UNIVERSIDAD NACIONAL DE COLOMBIA

Redes neuronales que expresan múltiples estrategias en el videojuego StarCraft 2

Miguel González Duque

Universidad Nacional de Colombia
Facultad de Ciencias, Escuela de Matemáticas
Medellín, Colombia
2019

Redes Neuronales que Expresan Múltiples Estrategias en el Videojuego StarCraft 2

Miguel González Duque

Tesis presentada como requisito parcial para optar al título de:
Magister en Ciencias - Matemáticas

Director:
Ph.D. Daniel Cabarcas Jaramillo

Línea de Investigación:
Inteligencia Artificial y Aprendizaje Automático

Universidad Nacional de Colombia
Facultad de Ciencias, Escuela de Matemáticas
Medellín, Colombia
2019

Neural Networks that Express Multiple Strategies in the Video Game StarCraft 2

Miguel González Duque

Universidad Nacional de Colombia
Facultad de Ciencias, Escuela de Matemáticas
Medellín, Colombia
2019

Arithmetical or algebraical calculations are, from their very nature, fixed and determinate. Certain "data" being given, certain results necessarily and inevitably follow. These results have dependence upon nothing, and are influenced by nothing but the "data" originally given. [...] But the case is widely different with the Chess-Player. With him there is no determinate progression. No one move in chess necessarily follows upon any one other. From no particular disposition of the men at one period of a game can we predicate their disposition at a different period.

- Edgar Allan Poe, *Maelzel's Chess-Player.*

To Beatriz González Alzate,
and to Daniel González Duque.

Acknowledgments

First of all, I would like to thank my supervisor Daniel Cabarcas Jaramillo. From day one, he supported this wild idea of dedicating a M.Sc. Thesis to Supervised Learning in StarCraft 2, and he had the patience to guide me through the process of elaborating this project and writing this thesis.

This thesis also wouldn't have been possible without the support and guidance of Niels Justesen and Sebastian Risi. I want to thank them for allowing me to visit their research group at the IT University of Copenhagen. This thesis is an extension of the work I did during my stay at ITU, collaborating with both authors to develop the approach this thesis studies.

During the development of this thesis, and during my M.Sc. in general, I have met wonderful people in Medellín and in Copenhagen. I want to thank my professors (Edgar Ramos, Camilo Arias, John Byron Baena and Juan Diego Vélez) and the members of the ML Seminar we started during my stay in Medellín, with a special mention to Santiago Pineda. The members of the Game AI group and the REAL Lab at ITU were also very welcoming during my stay in Copenhagen, and they played an important role in the development of these ideas (special thanks to Mads Lassen in this respect).

Finally, this thesis relies heavily on open source software. A big thanks to the developers and maintainers of python libraries like numpy, pandas, matplotlib, PyTorch and sci-kit learn.

Resumen

Usando redes neuronales y aprendizaje supervisado, hemos creado modelos capaces de solucionar problemas a nivel súperhumano. Sin embargo, el proceso de entrenamiento de estos modelos es tal que el resultado es una política que promedia todos los diferentes comportamientos presentes en el conjunto de datos. En esta tesis presentamos y estudiamos la técnica Aprendizaje por Imitación de Repertorios de Comportamiento (BRIL), la cual permite entrenar modelos que expresan múltiples comportamientos de forma ajustable. En BRIL, el usuario diseña un espacio de comportamientos, lo proyecta a bajas dimensiones y usa las coordenadas resultantes como entradas del modelo. Para poder expresar cierto comportamiento a la hora de desplegar la red, basta con fijar estas entradas a las coordenadas del respectivo comportamiento. La pregunta principal que investigamos es la relación entre el algoritmo de reducción de dimensionalidad y la capacidad de los modelos entrenados para replicar y expresar las estrategias representadas. Estudiamos tres algoritmos diferentes de reducción de dimensionalidad: Análisis de Componentes Principales (PCA), Mapeo de Características Isométrico (Isomap) y Aproximación y Proyección de Manifolds Uniformes (UMAP); diseñamos y proyectamos un espacio de comportamientos en el videojuego StarCraft 2, entrenamos diferentes modelos para cada embebimiento y probamos la capacidad de cada modelo de expresar múltiples estrategias. Los resultados muestran que, usando BRIL, logramos entrenar modelos que pueden expresar los múltiples comportamientos presentes en el conjunto de datos. La estructura geométrica preservada por cada método de reducción induce diferentes separaciones de los comportamientos, y estas separaciones se ven reflejadas en las conductas de los modelos.

Palabras clave: Aprendizaje Supervisado, Reducción de la dimensionalidad, Redes Neuronales, StarCraft 2, Aprendizaje por Imitación de Repetorios de Comportamiento.

Abstract

Using neural networks and supervised learning, we have created models capable of solving problems at a superhuman level. Nevertheless, this training process results in models that learn policies that average the plethora of behaviors usually found in datasets. In this thesis we present and study the Behavioral Repetoires Imitation Learning (BRIL) technique. In BRIL, the user designs a behavior space, the user then projects this behavior space into low coordinates and uses these coordinates as input to the model. Upon deployment, the user can adjust the model to express a behavior by specifying fixed coordinates for these inputs. The main research question ponders on the relationship between the Dimension Reduction algorithm and how much the trained models are able to replicate behaviors. We study three different Dimensionality Reduction algorithms: Principal Component Analysis (PCA), Isometric Feature Mapping (Isomap) and Uniform Manifold Approximation and Projection (UMAP); we design and embed a behavior space in the video game StarCraft 2, we train different models for each embedding and we test the ability of each model to express multiple strategies. Results show that with BRIL we are able to train models that are able to express the multiple behaviors present in the dataset. The geometric structure these methods preserve induce different separations of behaviors, and these separations are reflected in the models' conducts.

Keywords: Supervised Learning, Dimensionality Reduction, Neural Networks, StarCraft 2, Behavioral Repetoires Imitation Learning.

Contents

Introduction

Artificial Neural Networks (ANNs) are becoming the state-of-the-art model for solving many Supervised Learning (SL) and Reinforcement Learning (RL) problems. They are able to detect objects in images on a human-level [20], they are able to generate convincing text given a prompt [29], they can create convincing human faces [19], and they are able to solve a myriad of classification problems in medicine [10]. In the realm of video games, the combination of ANNs with Reinforcement Learning ideas has given rise to Deep Reinforcement Learning, a sub-branch that uses ANNs to approximate the key objects of Reinforcement Learning: Policies [35]. With these technologies, human-level game playing has been achieved in Chess, Shogi, Go [32, 33], a series of classic Atari games [25] and DoTA 2 [28].

Even though ANNs are a promising tool, a main limitant to their adoption in more environments is their inability to express multiple behaviors. Once trained, these models tend to overfit to a particular behavior or strategy when confronting the environment [17]. This is also the case in Supervised Learning, in which models learn an averaged behavior, limiting their expressiveness.

StarCraft II

The problem of training an ANN capable of adapting to multiple environments can be observed in complex Real-Time Strategy (RTS) video games. We chose **StarCraft 2** as our testbed since, in it, agents are able to express a plethora of different strategic behaviors.

In StarCraft II, two players face each other on a map and must employ strategic decision making in order to balance economy, army and research. The game wins when a player is able to destroy every building the opponent has. Each player can play in one of three races: Zerg, an alien race specially suited for quick attacks (or *rushes*) and whose units are inexpensive and numerous, but fragile; Protoss, an advanced alien race with expensive yet powerful units, and Terran, a balanced race. Figure **0-1** shows a screenshot of the game.

StarCraft II contains in it problems that were not tackled before in classic AI-game playing [2]:

- It is a game of imperfect information, since each player is not always able to see what the opponent is doing due to a fog-of-war mechanic.
- It is real-time, in the sense that both players are taking actions at the same time.

Figure 0-1.: A Replay Screenshot of StarCraft 2. This screenshot shows a Terran player attacking a Zerg player using Marines (the infantry), a Banshee (the flying unit with rotors), a Hellion (the car) and a Cyclone (the mechanic unit).

- Actions taken by the players sometimes have no immediate effect, which implies that agents must have long-term strategic planning.

The StarCraft saga has been the subject of plenty of research, specially when investigating the problem of creating human-level agents, or bots, and beating professional human players[1] [27]. This scientific interest shifted towards StarCraft II when DeepMind and Blizzard released the StarCraft II Learning Environment, an Application Programming Interface (API) designed to allow the community to perform Reinforcement Learning experiments [38].

Until 2019, the results were unsatisfactory, in the sense that the best performing bots were scripted agents that got easily defeated by good human players. In early 2019, DeepMind released preliminary results on AlphaStar, an agent that was able to beat two professional human players [37]. AlphaStar was trained using a combination of Supervised Learning and Reinforcement Learning. First, models were trained from human demonstrations, and then were set up as seeds in a tournament style Reinforcement Learning algorithm called the *AlphaStar League*, in which these models competed against one another until achieving human-level strength.

[1] StarCraft and StarCraft II are e-sports games, considered by many as the most challenging 1-vs-1 game in the world. To get a taste of how competitive and difficult the game is, we invite the reader to watch a match between the best Terran in the world and a Zerg player.

In this thesis, we deal with Supervised Learning of strategic decision making with the intention of making the bots more adaptive and able to express multiple behaviors. This has direct implications to both the problem of creating human-level agents in StarCraft II, and making better, more believable and fun bots to play against. One the one hand, these multiple behaviors learned by the model can be used to kick-start an *AlphaStar League*-like learning procedure and, on the other, human-level agents could be modulated to play a certain strategy.

Outside the realm of video games, our approach also has implications to circumstances in which agents are expected to have multiple behaviors, such as optimal control in search-and-rescue robotics, driver-less technology and text generation.

Contributions

In this thesis we study a novel method for creating more adaptive, more expressive ANN models called Behavioral Repertoires Imitation Learning (BRIL) [16], proposed by Justesen, Risi, and the author of this thesis. The idea of BRIL is as follows: we design a behavior space that encodes the different behaviors in a dataset using expert knowledge, we embed this behavior space in low dimensions using Unsupervised Learning techniques, and we use the coordinates of this low dimensional behavior space as new input nodes in the neural network.

In order to test BRIL we first replicate the results of [17] in StarCraft II, by training a neural network model from state-action pairs extracted from human games. This mas made possible by developing an open source tool, called `sc2reaper`[2], designed to allow its users to extract information from .SC2Replay files, which contain all the information related to a match of StarCraft II. This open source tool has been already used by the community in research projects [8].

We then apply BRIL on a new behavior space in StarCraft II. We study the impact of using different low-dimensional embedding algorithms on our novel approach. We trained models on these embeddings and tested their capacity to express different behaviors. After that, we coupled these models with a simple StarCraft 2 bot and we analyzed its behavior in-game. Finally, we learned the win probability for each model as an engine for the bot's strategy against a particular kind of opponent.

There is overlap between this thesis and the article that introduces BRIL [16]: both rely on the same extracted data, processed by the author of this thesis. Both include the same ideas and description of the BRIL methodology, discussed and proposed by Justesen, Risi and the author. Figure **3-1** is extracted from the article too. Nevertheless, the behavior spaces and their respective analysis are different and were computed independently by Justesen in the

[2] https://github.com/miguelgondu/sc2reaper

article and by the author in this thesis.

Thesis Layout

This thesis starts with two chapters that explain the mathematical background necessary to understand the approach. Chapter 1 is an introduction to Supervised Learning using Feed-Forward Neural Networks and Gaussian Processes and Chapter 2 explains 3 different Dimension Reduction techniques. After these chapters, the approach is presented on greater detail in Chapter 3; Chapter 4 explains the experiments that were carried out, their results and a discussion; finally, Chapter 5 summarizes and concludes the thesis. Appendix A contains a more computational-oriented description of the tools we used.

1. Supervised Learning

The idea of Supervised Learning is to try to infer the pattern in a training set $\mathcal{D} = \{(\boldsymbol{x}_n, \boldsymbol{t}_n)\}$. This means that we assume there is a functional form that is generating this data stochastically, that there is a function $f(\boldsymbol{x})$ that outputs $\boldsymbol{t}$. The goal, then, is to approximate this function f, knowing nothing but the training set $\mathcal{D}$. By approximating f, we will be able to predict target values $\boldsymbol{t}$ for previously unseen input values $\boldsymbol{x}$.

Supervised learning can be done using parametric and non-parametric models. In the parametric setting, we consider a function $y(\boldsymbol{x}, \boldsymbol{w})$ that depends on a set of adjustable parameters $\boldsymbol{w}$ that we modulate and tweak in order to fit y to the pattern f. In the non-parametric setting, we don't consider a closed parametric form for the approximating function y, but instead consider a family of functions, and start discarding those that do not explain our data.

In this chapter we introduce a special kind of parametric function called the Feed-Forward Neural Network (or FF-NN, for short). This function is a member of the more broader class of functions called Artificial Neural Networks (ANNs). Feed-Forward Neural Networks can be used as parametric models for solving Supervised Learning problems. We will also introduce a non-parametric tool for regression and classification called the Gaussian Process. The main references of this chapter are [1], [39], [30] and [12].

1.1. Introduction to neural networks

ANNs are a natural extension of Linear Regression and Classification. In Linear Regression, for example, the pattern behind a dataset is learned via the minimization of an error function in which linear combinations of adjustable parameters appear. These linear combinations are of the form $y(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w}^T \phi(\boldsymbol{x})$ where the $\boldsymbol{w} \in \mathbb{R}^M$ are adjustable weights and $\phi \colon \mathbb{R}^I \to \mathbb{R}^M$ is a vector of basis functions that extract so called *features* from the inputs. These basis functions where, in the past, defined manually using expert knowledge. The idea of neural networks is to introduce automatic feature extraction [21] by replacing these basis functions with linear combinations and non-linear activation functions. The amount of times this process is iterated turns into the amount of hidden layers of the network, and the multiple dimensions of feature spaces become the number of hidden nodes per layers.

In order to define FF-NNs, we will start with computational graphs (which allows us to decompose a complicated function in terms of its main operations). After that, we will define

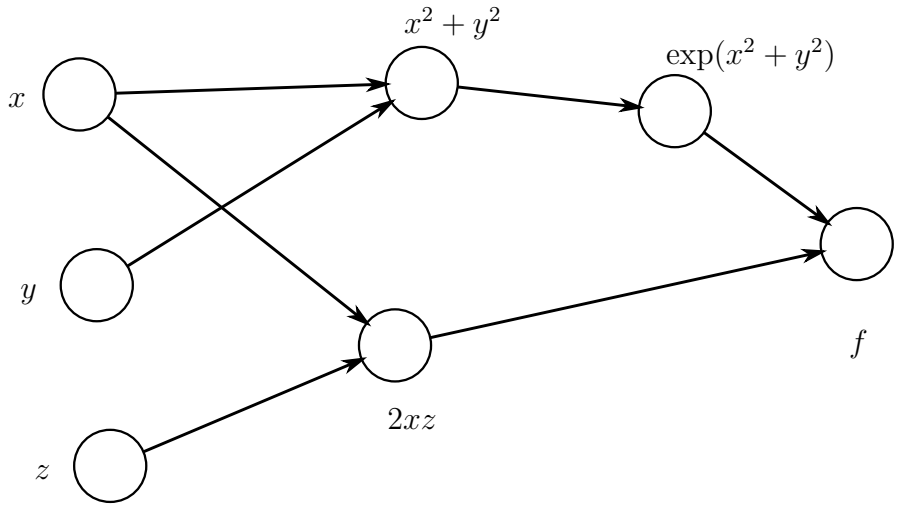

Figure 1-1.: A computational graph of function $f(x, y, z) = \exp(x^2 + y^2) + 2xz$.

these networks as a special type of computational graphs. Finally, we will use statistical inference to define the error function to optimize, and we will use an efficient algorithm for computing the gradients of this error function using the chain rule on computational graphs.

1.2. Computational graphs

Computational graphs let us describe arbitrary functions in terms of manipulating more basic operations. In this language, the chain rule and differentiation with respect to multiple variables becomes graphical.

More formally, we will define a **computational graph** as a directed weighted graph $G = (V, E, w)$ where V is a set of variables and functions. We can consider a computational graph as a function on a subset of input notes of V, and arbitrary functions can be decomposed into (possibly) multiple computational graphs.

As a simple example, consider the function $f(x, y, z) = \exp(x^2 + y^2) + 2xz$. A computational graph associated with this function f can be found in Figure 1-1.

The language of computational graphs allows us to easily define functions that would otherwise be difficult to write down. Moreover, given the nature of the nodes in between the inputs and the outputs, the chain rule can be easily computed in terms of paths in the computational graph. Indeed, the derivative of a node u with respect to another node v is the sum over all paths $P = (v = v_0, v_1, \ldots, v_n = u)$ of the product of all intermediate derivatives $\partial v_{i+1}/\partial v_i$. These intermediate nodes serve as **hidden variables**, as stepping stones in the differentiation of u with respect to v.

For example, the partial derivative with respect to x of the computational graph f, defined

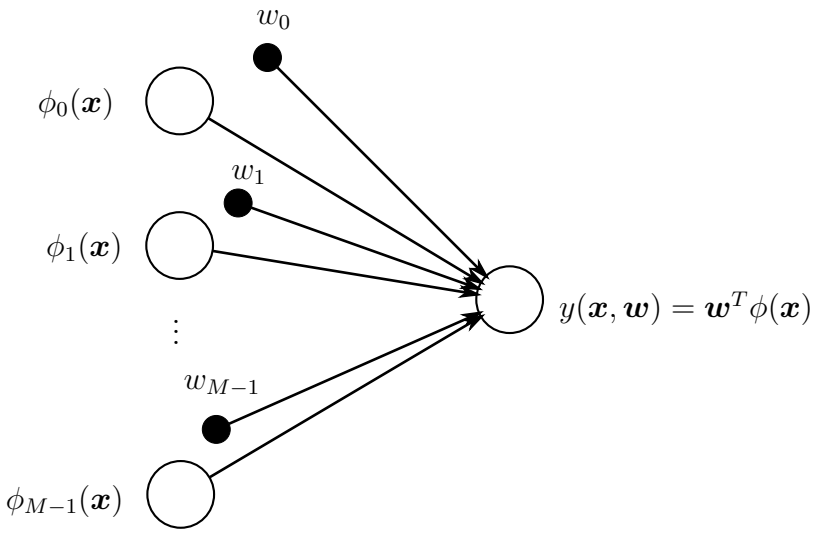

Figure 1-2.: The approximating function in linear regression $y(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w}^T\phi(\boldsymbol{x})$ as a computational graph. The weights $\boldsymbol{w} = (w_0, \ldots, w_{M-1})^T$ are represented in this graph as independent nodes in order to have a more transparent representation of how the derivatives in the error function will be computed from the graph.

in figure 1-1, is

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial \exp(x^2+y^2)} \frac{\partial \exp(x^2+y^2)}{\partial (x^2+y^2)} \frac{\partial (x^2+y^2)}{\partial x} + \frac{\partial f}{\partial (2xz)} \frac{\partial (2xz)}{\partial x} = 2x \exp(x^2+y^2) + 2z, \quad (1\text{-}1)$$

where the first term corresponds to the path between x and f going from x to x^2+y^2, to $\exp(x^2+y^2)$ and finally to f, and the second term corresponds to the path from x to f that goes through $2xz$.

1.3. Linear Regression as a Computational Graph

Before we introduce the concept of a FF-NN, let's state the problem of Linear Regression using a computational graph. First of all, Linear Regression consists on learning a set of weights $\boldsymbol{w} = (w_0, \ldots, w_{M-1})$ by fitting a function $y(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w}^T\phi(\boldsymbol{x})$ to a dataset of N supervised samples $\mathcal{D} = \{(\boldsymbol{x}_n, t_n)\}_{n=1}^N$. Figure 1-2 presents the computational graph of y.

In order optimize the parameters $\boldsymbol{w}$, we set an error function E that measures the dissimilarity between the target values t_n's and the predictions of the model $y(\boldsymbol{x}_n, \boldsymbol{w})$ and optimize $\boldsymbol{w}$ to minimize it. For this example, we will consider the **sum-of-squares** error function

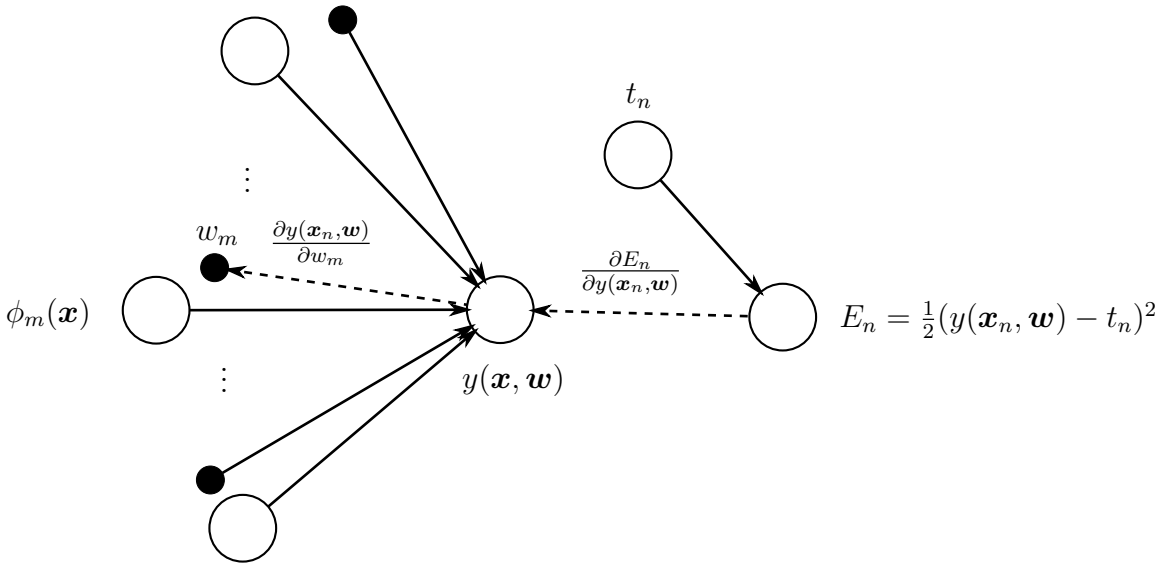

Figure 1-3.: The result of extending the computational graph of figure 1-2 by including a summand of the error function E_n, showing how to compute the derivative of E_n with respect to an arbitrary weight w_m.

given by

$$E(\mathcal{D}, \boldsymbol{w}) = \frac{1}{2}\sum_n (y(\boldsymbol{x}_n, \boldsymbol{w}) - t_n)^2. \tag{1-2}$$

We can add this to the computational graph of y in Figure 1-2 in order to easily determine the derivatives of E with respect to each w_m. Since differentiation is additive, we can focus on one of the summands of the error function, given by

$$E_n = \frac{1}{2}(y(\boldsymbol{x}_n, \boldsymbol{w}) - t_n)^2.$$

Figure 1-3 presents the expanded computational graph of the linear regression approximator y, and an example of a *derivative path* from one w_m to E_n. These are the inverted dashed lines. In summary, the derivative is given by

$$\frac{\partial E_n}{\partial w_m} = \frac{\partial E_n}{\partial y(\boldsymbol{x}_n, \boldsymbol{w})}\frac{\partial y(\boldsymbol{x}_n, \boldsymbol{w})}{\partial w_m} = (y(\boldsymbol{x}_n, \boldsymbol{w}) - t_n)\phi_m(\boldsymbol{x}_n), \tag{1-3}$$

and we could use an iterative optimization scheme such as gradient descent in order to optimize $E = \sum_n E_n$. We start with a random $\boldsymbol{w}^{(0)}$ and update it iteratively using

$$w_m^{(\tau+1)} = w_m^{(\tau)} - \eta \left(\frac{\partial E}{\partial w_m} \right) = w_m^{(\tau)} - \eta \left(\sum_n \boldsymbol{w}^{(\tau)T} \phi(\boldsymbol{x}_n) - t_n \right) x_m,$$

for some parameter $\eta > 0$ called the **learning rate**.

1.4. Definition and notation

An Artificial Neural Network is a special kind of a computational graph. The main class of Artificial Neural Networks we will be using in this thesis are FF-NNs, defined as follows: A computational graph $G = (V, E, w)$ is a FF-NN if:

(a) G is composed of $L+2$ layers of nodes where all nodes in layer l are fully connected with all nodes in layer $l+1$. The first layer is called the input layer, the last one is called the output layer, and the ones in between are called hidden layers.

(b) A node z in a hidden layer is computing a non-linear transformation of a weighted sum of all nodes in the previous layer v with their respective weights $w(v \to z)$, that is,

$$z = \sigma \left(\sum_{v \to z} w(v \to z) v \right)$$

for a transformation $\sigma \colon \mathbb{R} \to \mathbb{R}$ called an **activation function**, which could vary from layer to layer.

In the general case, ANNs allow for cycles and might be composed in different ways. For a survey on Deep Learning and the use and history of ANNs, we recommend [31].

We will denote the nodes at hidden layer l with $\boldsymbol{z}^{[l]}$, the input layer with $\boldsymbol{x}$ and the output layer with $\boldsymbol{y}$. Even though the weights are being considered as a function $w \colon E \to \mathbb{R}$, they are to be considered as variables, just like in the Linear Regression example.

Figure 1-4 presents a small example of a FF-NN with 2 hidden layers, 4 hidden nodes per layer, 3 inputs and 2 outputs. The linear regression function can be considered as a 0-hidden layer neural network (see Figure 1-2).

We can write down a simple formula for the communication between two consecutive layers. Suppose there are n_l nodes in layer l and n_{l+1} in layer $l+1$, then for some hidden node $z_j^{[l+1]}$ in layer $l+1$ we have

$$z_j^{[l+1]} = \sigma \left(\sum_{i=1}^{n_l} w \left(z_i^{[l]} \to z_j^{[l+1]} \right) z_i^{[l]} \right) \tag{1-4}$$

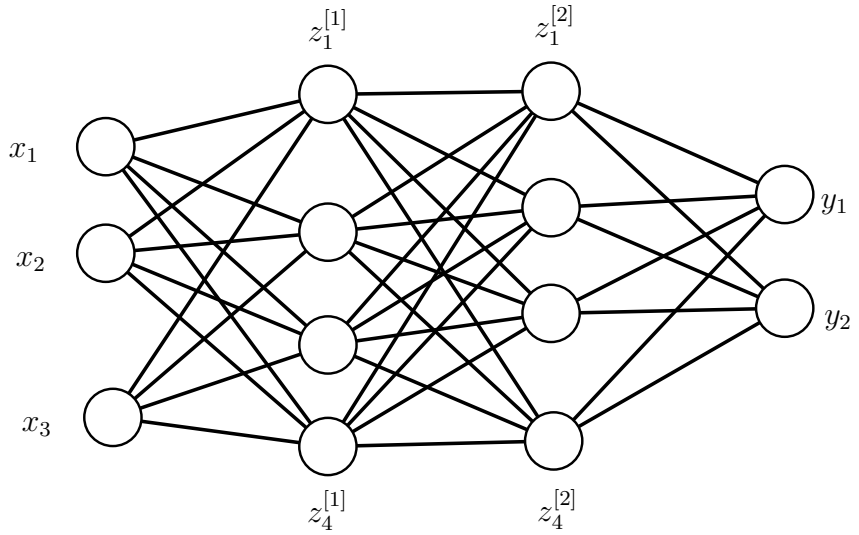

Figure 1-4.: A small example of a FF-NN with 2 hidden layers. We have omitted the arrows for simplicity.

Let's impose the notation $w\left(z_i^{[l]} \to z_j^{[l+1]}\right) =: w_{ji}^{[l]}$, then we have $z_j^{[l+1]} = \sigma\left(\boldsymbol{w}_j^{[l]T}\boldsymbol{z}^{[l]}\right)$ for $\boldsymbol{w}_j^{[l]} = \left(w_{j1}^{[l]}, w_{j2}^{[l]}, \ldots, w_{jn_l}^{[l]}\right)^T$. This notation allows us to introduce a matrix of weights. Indeed, we can easily compute $\boldsymbol{z}^{[l+1]} = \left(z_1^{[l+1]}, \ldots, z_{n_{l+1}}^{[l+1]}\right)$ as

$$\boldsymbol{z}^{[l+1]} = \begin{bmatrix} z_1^{[l+1]} \\ \vdots \\ z_{n_{l+1}}^{[l+1]} \end{bmatrix} = \sigma\left(\begin{bmatrix} \boldsymbol{w}_1^{[l]T}\boldsymbol{z}^{[l]} \\ \vdots \\ \boldsymbol{w}_{n_{l+1}}^{[l]T}\boldsymbol{z}^{[l]} \end{bmatrix}\right) = \sigma\left(W^{[l]T}\boldsymbol{z}^{[l]}\right), \tag{1-5}$$

where $W^{[l]}$ is the $n_l \times n_{l+1}$ matrix whose columns are given by $(\boldsymbol{w}_1^{[l]}, \ldots, \boldsymbol{w}_{n_{l+1}}^{[l]})$ and we have extended σ to act component-wise. In summary, the communication from one layer to the next is given by first taking a linear transformation and then passing through σ.

In most definitions of FF-NNs, an explicit independent term is considered in the communication between layers in Eq. (1-4), called the **bias**. We made the choice to not show it explicity for simplicity in the presentation, but feel free to always assume that there is a first node $z_0^{[l]} \equiv 1$ for all hidden layers. By adding this node, we are adding a constant $w_{j0}^{[l]}$ to every addition in Eq. (1-4).

1.5. The universal approximation theorem

Relatively simple FF-NNs are theoretically known to approximate any continuous function from a compact subset of $\mathbb{R}^n$ to $\mathbb{R}^m$, $n, m \in \mathbb{N}$. In 1988, Cybenko proved that a 1-hidden-

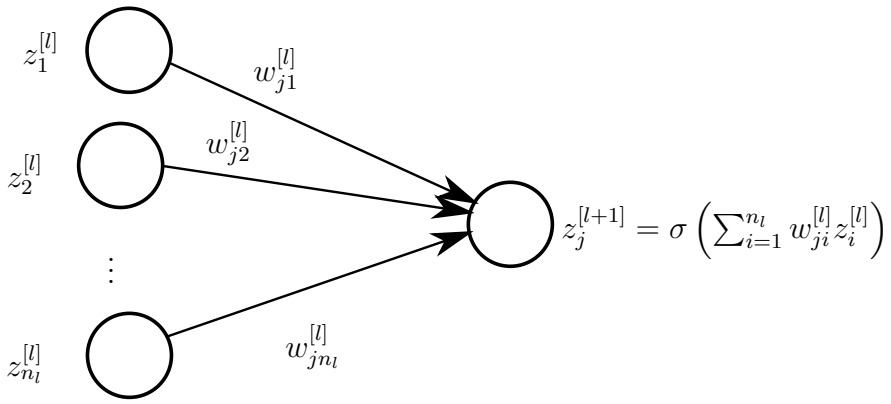

Figure 1-5.: The communication between one layer to the next. Every node on hidden layer $l+1$ is taking all the values computed by those in layer l and computing a weighted sum with parameters $\boldsymbol{w}_j^{[l]}$.

layer neural network with enough hidden nodes and an adequate activation function could approximate continuous functions as described above [6]. This theorem is known as **the universal approximation theorem**. Let $C(A)$ be the set of continuous functions from $A \subseteq \mathbb{R}^M$ to $\mathbb{R}$, for some $M \in \mathbb{N}$. In our notation, the universal approximation theorem goes like this:

Theorem 1.5.1. Let σ be a continuous function from $\mathbb{R}$ to $\mathbb{R}$ such that if μ is a measure in $[0,1]^M$,

$$\int_{[0,1]^M} \sigma(\boldsymbol{w}^T \boldsymbol{x} + w_0) d\mu(\boldsymbol{x}) = 0 \tag{1-6}$$

for all $\boldsymbol{w} \in \mathbb{R}^N$ and $w_0 \in \mathbb{R}$ implies that $\mu = 0$. Then finite sums of the form

$$G(\boldsymbol{x}) = \boldsymbol{w}^{[1]T} \left(\sigma\left(\boldsymbol{w}_j^{[0]T} \boldsymbol{x} + w_0\right)\right)_{j=1}^{n_0} \tag{1-7}$$

are dense in $C([0,1]^M)$, that is, for $\epsilon > 0$ and $f \in C([0,1]^M)$ there exists a sum of the form (1-7) such that

$$|G(\boldsymbol{x}) - f(\boldsymbol{x})| < \epsilon$$

for all $\boldsymbol{x} \in [0,1]^M$. The extension to approximating functions f from $\mathbb{R}^M$ to $\mathbb{R}^N$ is trivial.

The proof of this theorem is out of the scope of this thesis, but can be found in [6]. This theorem imposes restrictions on the activation function σ (namely, the hypothesis with respect to the Eq. (1-6)). Hornik et al. showed in 1989 that any continuous, non-constant,

non-decreasing and bounded function σ sufficed [15]. One example of them is the **sigmoid function** given by

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{1-8}$$

Another frequently used example of activation function is ReLU, given by the identity for non-negative values and constant 0 for negative ones. Even though ReLU doesn't follow the hypothesis of the universal approximation theorem, it has been proven that networks with ReLU activations can also approximate continuous functions between Euclidean spaces [34].

These activation functions are usually used on the hidden layers, since the activation function for the output layer must be selected according to the problem that is being considered (i.e. classification or regression).

1.6. Supervised learning using Neural Networks

As we stated in the beginning of the chapter, Supervised Learning consists on trying to find the pattern behind a training set, with the aim of generalizing to previously unseen inputs. The universal approximation theorem implies that neural networks could be used to solve this problem. In this section we will focus on how to use neural networks to solve the problem of classification.

In classification, we are given a training set $\mathcal{D} = \{\boldsymbol{x}_n, t_n\}_{n=1}^N$, where the t_n's are discrete labels in $\{1, \dots, C\}$ describing the class where each $\boldsymbol{x}_n$ belongs to. Solving the classification problem consists on finding a function that maps inputs $\boldsymbol{x}$ to one of these classes.

We will consider a 1-hot encoding of these classes, in which we will replace the discrete labels $t_n \in \{1, \dots, C\}$ for vectors $\boldsymbol{t}_n = (t_{n1}, \dots, t_{nC})$ where $t_{nc} = 1$ if the original t_n was c and 0 otherwise.

In order to solve the problem of classification, we assume that the data $\mathcal{D}$ is distributed according to the following generalized Bernoulli distribution:

$$p(\boldsymbol{t}|\boldsymbol{x}, \boldsymbol{w}) = \prod_{c=1}^{C} y_c(\boldsymbol{x}, \boldsymbol{w})^{t_{nc}}, \tag{1-9}$$

where $y_c(\boldsymbol{x}, \boldsymbol{w})$ is a parametric approximation of the probability that $\boldsymbol{x}$ belongs to class c. In order to find the parameters, we use **Maximum Likelihood approximation**. The Likelihood function is given by

$$L_{\mathcal{D}}(\boldsymbol{w}) = \prod_{n=1}^{N} \prod_{c=1}^{C} y_c(\boldsymbol{x}_n, \boldsymbol{w})^{t_{nc}}. \tag{1-10}$$

Notice that this Likelihood function is the joint distribution of our data, if we assume it to be identically and independently distributed with respect to Eq. (1-9). Instead of maximizing

it directly, we minimize the error function associated with it, which is defined as the negative log-likelihood:

$$E_{\mathcal{D}}(\boldsymbol{w}) = -\log(L_{\mathcal{D}}(\boldsymbol{w})) = -\sum_{n=1}^{N}\sum_{c=1}^{C} t_{nc}\log(y_c(\boldsymbol{x}_n, \boldsymbol{w})), \tag{1-11}$$

this error function is called the **cross-entropy** error function.

In order to approximate the class probabilities $y_c(\boldsymbol{x}, \boldsymbol{w})$, we use a FF-NN with a **softmax** output activation. Let $\boldsymbol{y}(\boldsymbol{x}, \boldsymbol{w})$ denote the output of a neural network with L hidden layers, then softmax is defined by

$$\boldsymbol{y}(\boldsymbol{x}, \boldsymbol{w}) = \text{softmax}(\boldsymbol{z}^{[L]}) := \left(\frac{\exp\left(z_c^{[L]}\right)}{\sum_{\tilde{c}=1}^{C}\exp\left(z_{\tilde{c}}^{[L]}\right)}\right)_{c=1}^{C}. \tag{1-12}$$

The softmax output is taking the last hidden layer and transforming it to a probability vector $\boldsymbol{y}(\boldsymbol{x}, \boldsymbol{w}) = (y_1(\boldsymbol{x}, \boldsymbol{w}), \ldots, y_C(\boldsymbol{x}, \boldsymbol{w}))$, that is, every component is between 0 and 1 and the sum of all components equals 1.

The output of this neural network can be considered a **policy** in the language of Reinforcement Learning. Since we will be dealing with video games (and agents playing them), it is worthwhile to properly define what a policy is:

Definition 1.6.1. Let S be the set of states an agent can be in, and let $A = \{a_1, \ldots, a_C\}$ be the set of actions the agent could take. A **policy** is a function $\pi\colon S \to \mathbb{R}^C$ that maps states $s \in S$ to probability vectors $\pi(s) = (\pi_1(s), \ldots, \pi_C(s))$ in which $\pi_c(s)$ is the probability (or level of confidence) that the agent would take action $a_c \in A$.

So, for the main problem of this thesis (described in Chapter 3), we will be using neural networks to learn policies, which is equivalent to solving classification problems, as can be seen from definition 1.6.1.

1.7. Backpropagation

We have now identified the error function that we ought to minimize in the problem of classification (or policy learning), that is, the cross-entropy defined by (1-11). What rests is describing an algorithm for efficiently computing the gradients of this error function with respect to the parameters in a FF-NN. Since this algorithm can be applied to any error function E, we state it in general terms. To set the notation, suppose we're given a training set $\mathcal{D} = \{\boldsymbol{x}_n, \boldsymbol{t}_n\}$, and that we have a neural network with L hidden layers.

Figure **1-6** shows the computational graph that results from connecting the error function to the output to the error function, and the paths that go from this error function to a

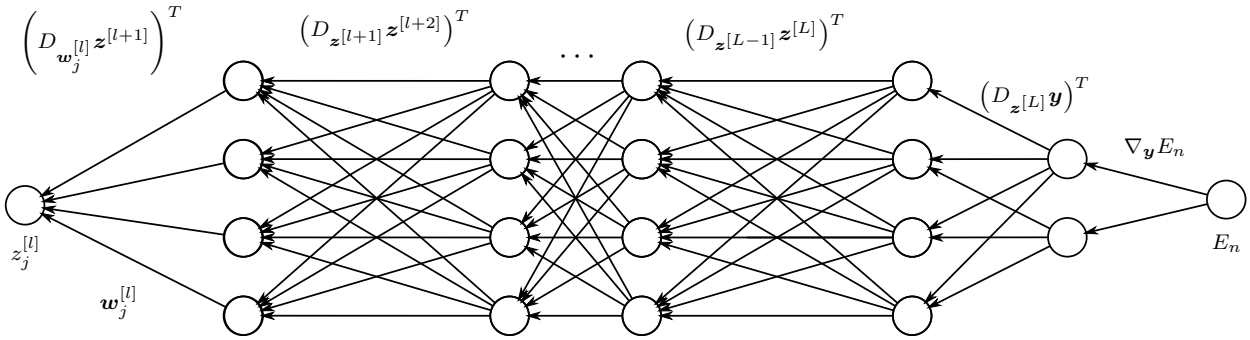

Figure 1-6.: In order to compute the gradient of E_n with respect to $\boldsymbol{w}_j^{[l]}$, we go layer by layer taking the respective Jacobian.

particular $w_{ji}^{[l]}$. In order to compute the partial derivative of E with respect to $w_{ji}^{[l]}$, we have to compute a series of gradients and Jacobians, according to the stepping stones between E and the lth layer:

$$\begin{aligned}\nabla_{\boldsymbol{w}_j^{[l]}}E &= \nabla_{\boldsymbol{w}_j^{[l]}}\left(E\circ\boldsymbol{y}\circ\boldsymbol{z}^{[L]}\circ\cdots\circ\boldsymbol{z}^{[l+1]}\right)\\ &= \left(D_{\boldsymbol{w}_j}\boldsymbol{z}^{[l+1]}\right)^T\left(D_{\boldsymbol{z}^{[l+1]}}\boldsymbol{z}^{[l+2]}\right)^T\cdots\left(D_{\boldsymbol{z}^{[L-1]}}\boldsymbol{z}^{[L]}\right)^T\left(D_{\boldsymbol{z}^{[L]}}\boldsymbol{y}\right)^T\left(\nabla_{\boldsymbol{y}}E\right),\end{aligned}$$

where by $D_{\boldsymbol{u}}\boldsymbol{v}$ we mean the Jacobian of $\boldsymbol{v}$ with respect to $\boldsymbol{u}$. We can compute these Jacobians: let i be between $l+1$ and L, then

$$\begin{aligned}D_{\boldsymbol{z}^{[i]}}\boldsymbol{z}^{[i+1]} &= D_{\boldsymbol{z}^{[i]}}\sigma\left(W^{[i]T}\boldsymbol{z}^{[i]}\right)\\ &= \left[\frac{\partial\sigma\left(W^{[i]T}\boldsymbol{z}^{[i]}\right)_r}{\partial z_s^{[i]}}\right]_{r,s}\\ &= \left[\frac{\partial\sigma\left(\boldsymbol{w}_r^{[i]T}\boldsymbol{z}^{[i]}\right)}{\partial z_s^{[i]}}\right]_{r,s}\\ &= \left[\sigma'\left(\boldsymbol{w}_r^{[i]T}\boldsymbol{z}^{[i]}\right)\frac{\partial(\boldsymbol{w}_r^{[i]T}\boldsymbol{z}^{[i]})}{\partial z_s^{[i]}}\right]_{r,s}\\ &= \left[\sigma'\left(\boldsymbol{w}_r^{[i]T}\boldsymbol{z}^{[i]}\right)w_{r,s}^{[i]}\right]_{r,s},\end{aligned}$$

which is almost the weight matrix for layer i.

If we have a pair $(\boldsymbol{x}_n,\boldsymbol{t}_n)\in\mathcal{D}$, we can compute these gradients $\nabla_{\boldsymbol{w}_j}E$ at $(\boldsymbol{x}_n,\boldsymbol{t}_n)$ after finding the values for the hidden nodes $z^{[1]}(\boldsymbol{x})$, $\boldsymbol{z}^{[2]}(\boldsymbol{z}^{[1]})$ and so on. This is called **forward propagation**. Once we have evaluated the values of all nodes in the computational graph, we can start computing the gradients by going backwards.

In summary, we have the following algorithm for computing the gradient contribution of a point to a vector of weights $\boldsymbol{w}_j^{[l]}$: given a point $(\boldsymbol{x}_n, \boldsymbol{t}_n)$,

1. Propagate $\boldsymbol{x}$ forward through the network, evaluating the nodes in all $L+2$ layers.
2. Evaluate the output and the error function E_n.
3. Start by computing the gradient of E_n with respect to the outputs $\boldsymbol{y}$, call it $B = \nabla_{\boldsymbol{y}} E_n(\boldsymbol{x}_n, \boldsymbol{t}_n)$
4. for i going from L down to $l+1$, do $B = (D_{\boldsymbol{z}^{[i-1]}} \boldsymbol{z}^{[i]})^T B$.
5. End by computing $B = (D_{\boldsymbol{w}_j^{[l]}} \boldsymbol{z}^{[l]})^T B$. This is the gradient of E_n with respect to $\boldsymbol{w}_j$.

This algorithm is known as **backpropagation**. In the first step, we are multiplying matrices using (1-5) $L+1$ times. If we let $n = \max\{n_0, \ldots, n_{L+1}\}$, each of this operation can be done in $O(n^\omega)$ (for some $\omega \in (2,3)$ using fast matrix multiplication [9]). Suppose the evaluation of E_n takes constant time. The fourth and fifth step make $O(L)$ iterations of a procedure that does $O(n^\omega)$ work again. In summary, the complexity of this version of backpropagation is $O(Ln^\omega)$.

The computational tool that we used for creating and training FF-NNs in our experiments was PyTorch[1].

1.8. A Primer on Gaussian Process Classification

In order to train a neural network using backpropagation, there needs to be a considerable amount of data in order to properly capture the pattern the data comes from. This is usually the case for parametric models. This begs the question: how could we solve the problem of classification when the amount of training points is scarce? In the rest of this chapter we will give a small introduction to Gaussian Processes, and how they can be used to solve the problem of binary classification when the number of training points is low. We will focus on computations and a high-level overview of the theory. For a more theoretically-oriented presentation, we recommend the main reference on the subject [30].

We define a Gaussian Process as a collection of random variables, such that every finite subset of them has joint Gaussian distribution. In the case of classification, we will assume we have a training set of the form $\mathcal{D} = \{(\boldsymbol{x}_n, t_n)\}_{n=1}^N$, and we will consider the random variables to be the values of the pattern t we are trying to learn from $\mathcal{D}$. The finite subsets of these continuous of variables are precisely the t_n's.

In essence, we can do regressions and classifications with Gaussian Processes by starting with a mean $m(\boldsymbol{x}) = \mathbb{E}[t(\boldsymbol{x})]$ and fixing a kernel function $k(\boldsymbol{x}, \boldsymbol{x}') = \mathbb{E}[(t(\boldsymbol{x}) - m(\boldsymbol{x}))(t(\boldsymbol{x}') - m(\boldsymbol{x}'))]$, and we will update the mean m according to the data in $\mathcal{D}$.

Kernel functions are designed to measure the covariance between the outputs of two points $\boldsymbol{x}, \boldsymbol{x}'$. Thus, points with similar outputs are expected to get a number close to 1 and points

[1] https://pytorch.org/

with dissimilar outputs are expected to get a number close to 0. Notice that, by their definition above, they must be symmetric and positive semi-definite. An example of a kernel function is the **Radial Basis Function** of lengthscale l, or RBF, given by

$$k_l(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2l}\|\boldsymbol{x} - \boldsymbol{x}'\|^2\right), \tag{1-13}$$

where the metric is Euclidean.

For simplicity, we will start by assuming that the targets are jointly Gaussian with mean 0. Consider the problem of predicting a new value t_* for a new test input $\boldsymbol{x}_*$. Since the outputs are jointly Gaussian, we have

$$\begin{bmatrix} t_1 \\ \vdots \\ t_N \\ t_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X,X) & K(X,\boldsymbol{x}_*) \\ K(\boldsymbol{x}_*,X) & k(\boldsymbol{x}_*,\boldsymbol{x}_*) \end{bmatrix}\right), \tag{1-14}$$

where $K(X,X)$ is the matrix whose (i,j)th component is given by $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, $K(X,\boldsymbol{x}_*)$ is the vector with components $k(\boldsymbol{x}_1,\boldsymbol{x}_*),\ldots,k(\boldsymbol{x}_N,\boldsymbol{x}_*)$ and $K(\boldsymbol{x}_*,X) = K(X,\boldsymbol{x}_*)^T$. In order to get a predictive distribution on t_*, we only need to condition on the seen targets $t_1,\ldots,t_N$. We can compute the distribution on t_* using the definition of conditional probability:

$$\begin{aligned} p(t_*|\boldsymbol{x}_*, \mathcal{D}) = p(t_*|K(\boldsymbol{x}_*,X)K(X,X)^{-1}\boldsymbol{t}, \\ k(\boldsymbol{x},\boldsymbol{x}) - K(\boldsymbol{x}_*,X)K(X,X)^{-1}K(X,\boldsymbol{x}_*)). \end{aligned} \tag{1-15}$$

With Eq. (1-15), we are able to solve the regression problem: after updating the mean and variance with the training data $\mathcal{D}$, we can not only predict the value of new outputs t_* given $\boldsymbol{x}_*$, but also measure how much certainty we have in that value using the variance.

After knowing how to perform regression with Gaussian Processes, we can use it to solve the problem of binary classification in a way that is analogous to how Logistic Regression uses Linear Regression. In this setting, we are trying to learn the probability that an input $\boldsymbol{x}$ belongs to one of the classes $\{0,1\}$,

$$\pi(\boldsymbol{x}) = p(t = 1|\,\boldsymbol{x}).$$

To do so, we start by introducing a *nuisence* variable $f(\boldsymbol{x})$, and then transform its output to the $[0,1]$ interval using the sigmoid activation function (see Eq. 1-8). We perform a Gaussian Process Regression over f, but only focus on our prediction of the class probability

$$\pi(\boldsymbol{x}) = p(t = 1|\,\boldsymbol{x}) = \sigma(f(\boldsymbol{x})).$$

Unfortunately, this non-linear transformation may break the Gaussian behavior of the integrals that appear when marginalizing and predicting, making them analytically untractable. Numerical approximations of these integrals (either based on Monte Carlo sampling or on fitting the distributions with Gaussians) have been introduced, buy lay outside of the scope of this thesis. For further reading, we heavily recommend [30].

1.9. Summary

In this chapter we introduced Supervised Learning as the problem of determining a functional relationship between pairs of supervised training points $\mathcal{D} = \{(\boldsymbol{x}_n, \boldsymbol{t}_n)\}_{n=1}^{N}$, with the aim of getting a predictive model for previously unseen input values.

We defined FF-NNs as a special kind of computational graph, and we have seen that we can infer a set of parameters $\boldsymbol{w}$ to fit a neural network to the pattern a training data set is coming from. In order to do so, we assume a distribution of the data and use Maximum Likelihood estimators to define an error function. We compute the gradients of this error function with respect to the parameters using an algorithm called backpropagation, which is essentially the chain rule.

Finally, we presented a brief summary of how Gaussian Processes are defined and how they can be used in order to solve regression and classification problems. A Gaussian Process consists on a set of random variables, any finite number of which are jointly Gaussian. In this setting, predicting new values can be done by conditioning the distribution of new data on $\mathcal{D}$. thus allowing only for models that already explain our data.

2. Dimensionality Reduction

Dimension reduction techniques aim at finding a low dimensional representation of high dimensional data. This low dimensional embedding of the data is expected to preserve as much of the original *structure* of the data as possible. Since this notion of structure is rather vague, each method aims to preserve certain specific properties of the original data. In this chapter we will study three algorithms for dimensionality reduction:

- Principal Component Analysis (PCA), which embeds the high-dimensional data into a linear subspace such that the variance after the embedding is maximized [1, 13].
- Complete Isometric Feature Mapping (Isomap), which creates a weighted graph that is used to approximate the geodesic distance of the low dimensional submanifold in which the data is assumed to lie. After creating this graph, an embedding is created from the distance matrix using Multidimensional Scaling (MDS), another dimension reduction algorithm [36, 41].
- Uniform Manifold Approximation and Projection (UMAP), which also assumes that the data is lying in a low dimensional submanifold of the ambient space and aims to reconstruct an embedding of it in low dimensions using tools from Riemannian Geometry and Algebraic Topology which can be thought of as manipulation of graphs [24].

As we will see in this chapter, all dimension reduction techniques are trying to approximate or learn a manifold in which the data is assumed to lie. We will introduce a terminology that will be used repeatedly in the following sections (especially those of Isomap and UMAP). These definitions will not be fundamental in the end, since Isomap and UMAP can be framed as algorithms that embed a graph into low dimensional space using an optimization procedure. Therefore, the reader can feel free to skip this section and go directly into the description of the methods if they have no interest in the geometric definitions that constitute the foundations of Isomap and UMAP. For further reading, we recommend [23, 7].

This chapter assumes some familiarity with the definition of a manifold M as a topological space that is Hausdorff, second-contable and locally Euclidean, that is, around every point $p \in M$ there exists an open set U and a homeomorphism that sends U to an open set of some $\mathbb{R}^L$, for some $L \in \mathbb{N}$.

Moreover, we specifically deal with the case of Riemannian Manifolds (M, g), in which g is pointwise and locally an inner product of tangent vectors to the manifold. This means that, for every $p \in M$, we have an inner product g_p in T_pM, the tangent space of M at p. This

inner product allows us to measure distances in M. These intrinsic distances that g allows are called **geodesic distances** in the manifold.

Isomap and UMAP can be thought of as instances of Manifold Learning [11, 3], in which the main hypothesis is that high-dimensional data must lie on or near a low dimensional manifold, and the goal is to learn the structure of this embedded manifold. The algorithms that we present here (except for PCA) do not learn a mapping from the high-dimensional space to the lower dimensional one, but only a set of vectors that in some way represent the data. New developments consider the problem of learning this mapping using Variational Autoencoders (a type of ANN) and Bayesian Techniques [14].

This chapter will start with introductions to PCA, Isomap, t-SNE and UMAP, and it ends with a description of K-means clustering, which aims at dividing the low dimensional representation into K groups, whose points are characterized for being close to one another.

In all these sections, we start with a set of points $\mathcal{D} = \{\boldsymbol{x}_n\}_{n=1}^N$ in high-dimension $\mathbb{R}^H$ (H for "high").

2.1. PCA

The idea behind PCA is to project the set of points $\mathcal{D}$ into a linear subspace generated by the directions in which the variance is maximized after the embedding. These directions are called **principal components**, and are expected to form an orthonormal basis of the linear subspace. Intuitively speaking, by maximizing the amount of variance in the data we are projecting to the space in the direction in which the data is more spread out and, thus, we are losing the least amount of information.

Mathematically speaking, we are trying to find vectors $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_L$ in $\mathbb{R}^H$ (where $L < H$) such that after projecting the points in $\mathcal{D}$ to the subspace spanned by $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_L$, their variance is maximized. Recall that the projection of a point $\boldsymbol{x}_n$ into the span of a single $\boldsymbol{u}_l$ is given by $\boldsymbol{u}_l^T \boldsymbol{x}$. Thus, the variance we are trying to maximize is

$$
\begin{aligned}
\frac{1}{N}\sum_{n=1}^{N}\left(\boldsymbol{u}_l^T\boldsymbol{x}_n - \boldsymbol{u}_l^T\overline{\boldsymbol{x}}\right)^2 &= \frac{1}{N}\sum_{n=1}^{N}\left(\boldsymbol{u}_l^T(\boldsymbol{x}_n - \overline{\boldsymbol{x}})\right)^2 \\
&= \frac{1}{N}\sum_{n=1}^{N}\boldsymbol{u}_l^T(\boldsymbol{x}_n - \overline{\boldsymbol{x}})(\boldsymbol{x}_n - \overline{\boldsymbol{x}})^T\boldsymbol{u}_l \\
&= \boldsymbol{u}_l^T\frac{1}{N}\sum_{n=1}^{N}\left[(\boldsymbol{x}_n - \overline{\boldsymbol{x}})(\boldsymbol{x}_n - \overline{\boldsymbol{x}})^T\right]\boldsymbol{u}_l \\
&= \boldsymbol{u}_l^T\Sigma\boldsymbol{u}_l,
\end{aligned}
$$

where $\overline{\boldsymbol{x}}$ is the mean of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$, and Σ is the covariance matrix, given by

$$\Sigma = \frac{1}{N} \sum_{n=1}^{N} \left[(\boldsymbol{x}_n - \overline{\boldsymbol{x}})(\boldsymbol{x}_n - \overline{\boldsymbol{x}})^T \right]. \tag{2-1}$$

Since the base of principal components is expected to be orthonormal (i.e. $\boldsymbol{u}_l^T \boldsymbol{u}_l = 1$), we can add a Lagrange Multiplier λ_l to our objective function $\boldsymbol{u}_l^T \Sigma \boldsymbol{u}_l$ forming a constrained optimization problem. This problem is solved by taking the gradient of

$$\boldsymbol{u}_l^T \Sigma \boldsymbol{u}_l - \lambda_l (1 - \boldsymbol{u}_l^T \boldsymbol{u}_l), \tag{2-2}$$

which is given by

$$\nabla_{\boldsymbol{u}_l} \left(\boldsymbol{u}_l^T \Sigma \boldsymbol{u}_l - \lambda_l (1 - \boldsymbol{u}_l^T \boldsymbol{u}_l) \right) = 2\Sigma \boldsymbol{u}_l - 2\lambda_l \boldsymbol{u}_l, \tag{2-3}$$

and after setting this gradient to be 0, we arrive at the following condition for $\boldsymbol{u}_l$:

$$\Sigma \boldsymbol{u}_l = \lambda_l \boldsymbol{u}_l, \tag{2-4}$$

which means that the eigenvalues of Σ are the critical points of the variance, and since the objective function is a quadratic form, these must be where the variance is maximized.

Moreover, since we were expecting the directions to be orthonormal, we can easily get an expression for the variance: left-multiplying equation (2-4) by $\boldsymbol{u}_l^T$, we get that the eigenvalue λ_l is precisely the variance after projecting to the span of eigenvector $\boldsymbol{u}_l$:

$$\boldsymbol{u}_l^T \Sigma \boldsymbol{u}_l = \lambda_l. \tag{2-5}$$

These computations motivate the following algorithm:

1. Compute the covariance matrix Σ of $\{\boldsymbol{x}_n\}_{n=1}^N$, given by (2-1).
2. Compute the eigenvectors and eigenvalues of Σ and sort them according to their eigenvalues in decreasing order, arriving at $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_H$ where $\lambda_1 > \lambda_2 > \cdots > \lambda_H$.[1]
3. Select the first L eigenvectors, and project $\{\boldsymbol{x}_n\}_{n=1}^N$ to the space generated by these directions.

The total variance is given by the sum of all eigenvalues. We can compute the percentage of the variance we are retaining with projecting to the first L eigenvalues by taking the ratio

$$\frac{\sum_{l=1}^{L} \lambda_l}{\sum_{l'=1}^{H} \lambda_{l'}}. \tag{2-6}$$

The algorithm we presented is $O(NH^2 + H^3)$ in complexity, since computing the covariance matrix is $O(NH^2)$ and the complexity of finding its eigenvalue decomposition is $O(H^3)$.

[1] All eigenvalues are different and real since Σ is symmetric and thus the eigenvectors form an orthonormal basis (under certain assumptions over Σ, which we assume to be true).

PCA is usually presented in another fashion: if we shift the vectors so as to have 0 mean, the covariance matrix takes a form that is proportional to $X^T X$, if X is the matrix whose columns are the data points $\{\boldsymbol{x}_n\}_{n=1}^N$. Since the eigenvalues only shift by a constant proportion, we could work with $X^T X$ in the case $\overline{\boldsymbol{x}} = 0$.

For our experiments, we used the scikit-learn implementation of PCA.

2.2. Isomap and MDS

The second Dimension Reduction algorithm that we employed in our experiments is Isomap, which consists of a smart application of Multidimensional Scaling (MDS). In this section we will explain both algorithms, starting with MDS.

2.2.1. Multidimensional Scaling

Multidimensional Scaling takes as input a distance matrix $D = [d_{ij}]_{N\times N}$ and tries to find points $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N$ in a space $\mathbb{R}^L$ such that the distance between $\boldsymbol{z}_i$ and $\boldsymbol{z}_j$ is d_{ij}. In the case of using the Euclidean distance in $\mathbb{R}^L$, there is an exact solution to this problem. For the sake of completeness, we construct said solution.

Assume without loss of generality that the points $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N$ all add up to 0. Consider the matrix $D^{\circ 2} = D \circ D = [d_{ij}^2]_{N\times N}$ (where $\circ$ denotes component-wise multiplication), since we are using the Euclidean distance,

$$d(\boldsymbol{z}_i, \boldsymbol{z}_j)^2 = \boldsymbol{z}_i^T \boldsymbol{z}_i - 2\boldsymbol{z}_i^T \boldsymbol{z}_j + \boldsymbol{z}_j^T \boldsymbol{z}_j, \tag{2-7}$$

which means that, if we call Z the $L \times N$ matrix whose columns are $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N$, $\mathbf{1}$ the $N \times 1$ vector with 1 in every component and $\boldsymbol{\psi} = [\boldsymbol{z}_n^T \boldsymbol{z}_n]_{n=1}^N$, we can write the following expression for $D^{\circ 2}$:

$$D^{\circ 2} = \boldsymbol{\psi}\mathbf{1}^T - 2Z^T Z + \mathbf{1}\boldsymbol{\psi}^T. \tag{2-8}$$

We are tasked to finding a way of solving for Z. After centering $D^{\circ 2}$ and recalling that all $\boldsymbol{z}_1 + \cdots + \boldsymbol{z}_N = 0$, we can recover

$$Z^T Z = -\frac{1}{2}\left(I - \frac{1}{N}\mathbf{1}\mathbf{1}^T\right) D^{\circ 2} \left(I - \frac{1}{N}\mathbf{1}\mathbf{1}^T\right). \tag{2-9}$$

Since $Z^T Z$ is symmetric, it admits an eigendecomposition

$$Z^T Z = U \operatorname{diag}(\lambda_1, \ldots, \lambda_L) U^T, \tag{2-10}$$

for some U of size $N \times L$. This implies that we can recover $Z = \operatorname{diag}(\lambda_1^{1/2}, \ldots, \lambda_L^{1/2}) U^T$. This line of reasoning allows us to consider the following algorithm for MDS:

1. Given the distance matrix $D = [d_{ij}]_{i,j}^N$ that the embedded points are assumed to obey, compute $D^{\circ 2} = [d_{ij}^2]$.
2. Find the eigendecomposition of

$$H = -\frac{1}{2}\left(I - \frac{1}{N}\mathbf{1}\mathbf{1}^T\right) D^{\circ 2} \left(I - \frac{1}{N}\mathbf{1}\mathbf{1}^T\right), \tag{2-11}$$

   arriving at a set of eigenvalues $\lambda_1, \ldots, \lambda_L$ and a basis change matrix $U \in \mathbb{N} \times \mathbb{L}$.
3. The vectors $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N$ in $\mathbb{R}^L$ are given by the columns of

$$\operatorname{diag}\left(\sqrt{\lambda_1}, \ldots, \sqrt{\lambda_L}\right) U^T.$$

Let's analyze the complexity of this algorithm: computing $D^{\circ 2}$ is $O(N^2)$, and finding the eigenvalue decomposition of matrix H in Eq. 2-11 is $O(N^3)$, which means that the overall complexity of MDS is $O(N^3)$.

In some other cases (where it makes more sense to consider another distance, different from the Euclidean), there exists a non-metric version of MDS. In it, we could find the position of these $\boldsymbol{z}_n$'s by minimizing a **stress function**, given by

$$S_{\mathcal{D}}(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N) = \sum_{i \neq j} (d_{ij} - \|\boldsymbol{z}_i - \boldsymbol{z}_j\|)^2. \tag{2-12}$$

Variants of this stress function give rise to alternative methods, such as e.g. Sammon Mapping [13].

2.2.2. Isomap

Isomap tries to embed data in low dimensions such that the global distance structure of the manifold is preserved, that is, such that points that are far away in the manifold (in terms of the geodesic distance) will stay far away after embedding them.

In order to preserve this global structure, Isomap starts by approximating geodesic distances in the manifold with the metric of the ambient manifold $\mathbb{R}^H$ by considering only small open neighborhoods around the data points. This means that, if we construct a graph by joining each point $\boldsymbol{x}_n$ with all of its k nearest neighbors (k being a hyperparameter to tweak) and weighting these edges with the distance in ambient space, we could construct a graph G and approximate all pairs of geodesic distances in the manifold with an all-pairs shortest paths algorithm in G. Once we have the matrix of all-pairs distances in the graph, we can feed it through MDS in order to get a low dimensional embedding.

In summary, Isomap consists of the following algorithm: Given a set of points $\{\boldsymbol{x}_n\}_{n=1}^N$ in $\mathbb{R}^H$, a positive integer $L < H$ and a hyperparameter k,

1. Create a graph G with $V = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ by joining each $\boldsymbol{x}_i$ with its k nearest neighbors. The weights of these edges are given by the distances between them in $\mathbb{R}^H$.

2. In this graph, compute the matrix of all-pairs shortest paths D.
3. Use MDS to find a low dimensional representation of this matrix distance D.

The complexity of the first step in this algorithm is $O(HN^2 + kN^2)$, since we are finding the k nearest neighbours for all points. Depending on the all-pairs shortests paths algorithm used in the second step, the complexities might vary. The most common one is Floyd-(Roy-Kleene-)Warshall's algorithm, which is $\Theta(N^3)$ [9].

For Isomap to properly infer the global distance structure of the manifold, it is assuming that the data points $\mathcal{D}$ were lying uniformly around the manifold. This does not have to be the case in general, and the next algorithm we will review tackles this problem in particular.

We also used scikit-learn's implementation of Isomap for our experiments.

2.3. UMAP

UMAP takes the idea of approximating the geodesic distance of the data manifold locally and cements it with proper theoretical justifications. This method consists of three steps: the geodesic distances of the data Manifold are approximated using tools from differential geometry, constructing several incompatible discrete metric spaces. Secondly, these discrete metric spaces are glued together to form a topological representation of the data manifold using fuzzy simplicial sets. Finally, this topological representation is embedded by minimization of a cross-entropy between the topological representation and a low dimensional one. In this section we will dive into the details of the approximation of geodesic distances (since it shares ideas with Isomap), and we will tackle the construction of the topological representation at a high level.

Fortunately, this algorithm can finally be enunciated only in terms of manipulation of graphs. Even if the parameters that appear will look out-of-the-blue after skipping the topological constructions, we can assure the reader that these decisions were based on proper mathematical foundations. Our presentation will follow the seminal paper by McInnes, [24], and all the missing theoretical details can be found in it.

2.3.1. Geodesic Approximation

Just like in Isomap, we want to approximate the geodesic distance in the manifold the data is assumed to lie on. The following theorem says that if the manifold is locally "flat enough" (that is, the distance is almost the Euclidean distance, a constant diagonal matrix), then we can locally approximate the geodesic distance with the metric of the ambient space, after a normalization.

Theorem 2.3.1. Let (M, g) be a Riemannian Manifold in an ambient $\mathbb{R}^H$ and let $p \in M$. If g is locally constant about p in an open neighborhood U such that g is a constant

diagonal matrix in ambient coordinates, then in a ball $B \subseteq U$ centered at p with volume $\pi^{H/2}/\Gamma(H/2+1)$ with respect to g, the geodesic distance from P to any point $q \in B$ is $(1/r)d_{\mathbb{R}^H}(p,q)$, where r is the radius of the ball in ambient space and $d_{\mathbb{R}^H}$ is the existing metric on the ambient space.

Notice that this theorem allows us to approximate the geodesic distance from a data point $\boldsymbol{x}_n$ to its neighbors using k nearest neighbors if we assume the data to be uniformly distributed (just like Isomap). Indeed, if the points are evenly distributed on the manifold, each ball of constant volume around any $\boldsymbol{x}_n$ will contain approximately the same number of points, and given a $\boldsymbol{x}_n$, the ball that contains its k-nearest neighbors is approximately of the same volume, regardless of $n \in \{1, \dots, N\}$.

UMAP starts by creating several metric spaces, one for each point in $\mathcal{D}$, in such a way that the hypothesis of uniform distribution is satisfied. This family of (extended) pseudo-metric spaces[2] $\{(\mathcal{D}, d_n)\}_{n=1}^N$ which obey the condition of uniform distribution are then patched together to form a topological representation.

2.3.2. Topological representation

In order to build a consistent topological representation from these inconsistent distances, UMAP takes advantage of the functorial properties of the categories of finite extended-pseudo-metric spaces and finite fuzzy simplicial sets (see [24] for definitions and technical details). At a high level, this means that we can translate the finite metric spaces we constructed in the first step to the more operational, more combinatorial fuzzy simplicial sets. We can join these fuzzy simplicial sets using a fuzzy union to create the topological representation of the data $\mathcal{X}$.

In order to embed this topological representation, UMAP starts with an embedding (say, placing N points randomly or running PCA) and then optimizes it, so as to have a topological representation close to that found via the first step.

2.3.3. Optimization of the low dimensional representation

Suppose we have a low dimensional representation $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N \in \mathbb{R}^L$ of $\mathcal{D}$. Since this representation can be though of as being in the manifold $\mathbb{R}^L$ with metric $d_{\mathbb{R}^L}$, we can follow the same computations we did for the high dimensional data in order to arrive at a fuzzy topological representation denoted by $\mathcal{Y}$.

UMAP ends with optimizing the low dimensional embedding $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$ so as to make $\mathcal{Y}$ as close as possible to $\mathcal{X}$ (the topological representation of the high dimensional data). Recall

[2] A **extended pseudo-metric space** is given by a set X and a distance d that satisfies $d(x,y) \geq 0$, $d(x,x) = 0$ and $d(x,z) \leq d(x,y) + d(y,z)$ or $d(x,z)$. That is, we allow for infinite distances.

that both $\mathcal{Y}$ and $\mathcal{X}$ are fuzzy sets. A measure of dissimilarity between fuzzy sets is the **fuzzy set cross-entropy**, which is very similar to the multi-class cross entropy considered in Chapter 1, (Eq. 1-11). Given two fuzzy sets from the same domain $\nu\colon A \to [0,1]$ and $\eta\colon A \to [0,1]$, we define the fuzzy set cross entropy as

$$C(\nu,\eta) = \sum_{a\in A}\left(\nu(a)\log\left(\frac{\nu(a)}{\eta(a)}\right) + (1-\nu(a))\log\left(\frac{1-\nu(a)}{1-\eta(a)}\right)\right). \tag{2-13}$$

We can then try to minimize $C(\mathcal{X},\mathcal{Y})$ with respect to the positions of the low dimensional representation $\boldsymbol{y}_1,\ldots,\boldsymbol{y}_N$. This can be done with gradient descent algorithms.

2.3.4. UMAP as a graph embedding algorithm

Just like Isomap, UMAP can be stated in terms of manipulation of graphs. In the case of UMAP, a weighted graph G is computed and then it is embedded into low dimensional space by an iterative scheme.

In UMAP, we construct a graph G as follows. Start by creating a graph $\overline{G}$ with $\mathcal{D} = \{\boldsymbol{x}_n\}_{n=1}^N$ as the vertices, and join each $\boldsymbol{x}_n$ with its k nearest neighbors $\{\boldsymbol{x}_{n_1},\ldots,\boldsymbol{x}_{n_k}\}$. Let ρ_n be the distance from $\boldsymbol{x}_n$ to its nearest neighbor and define σ_n as the real value that satisfies

$$\sum_{j=1}^{k}\exp\left(\frac{-\max(0,d(\boldsymbol{x}_n,\boldsymbol{x}_{n_j})-\rho_n)}{\sigma_n}\right) = \log_2(k).$$

The motivation for this selection of ρ_n and σ_n is buried in the technical details behind the definitions of the functors between finite metric spaces and fuzzy simplicial sets. After finding this σ_n, weight every edge from $\boldsymbol{x}_n$ to $\boldsymbol{x}_{n_j}$ by

$$w(\boldsymbol{x}_n,\boldsymbol{x}_{n_j}) = \exp\left(\frac{-\max(0,d(\boldsymbol{x}_n,\boldsymbol{x}_{n_j})-\rho_n)}{\sigma_n}\right).$$

With this, we have constructed a weighted graph $\overline{G}$. Let A be the adjacency matrix of $\overline{G}$. The graph G that UMAP later embeds into lower dimensions is constructed so as to have the following matrix as its adjacency matrix:

$$B = A + A^T - A\circ A^T,$$

where by $\circ$ we mean component-wise multiplication.

The numerical minimization of equation (2-13) translates into attractive and repulsive forces in the vertices of graph G. In particular, UMAP applies rounds of these attractive and repulsive forces iteratively until convergence (that is, until finding a minimum of the fuzzy set cross entropy). The attractive force between $\boldsymbol{y}_i$ and $\boldsymbol{y}_j$ is given by

$$\frac{-2ab\|\boldsymbol{y}_i-\boldsymbol{y}_j\|^{2(b-1)}}{1+\|\boldsymbol{y}_i-\boldsymbol{y}_j\|}w(\boldsymbol{x}_i,\boldsymbol{x}_j)(\boldsymbol{y}_i-\boldsymbol{y}_j),$$

where a and b are hyperparameters and the distance is Euclidean. The repulsive force between $\boldsymbol{y}_i$ and $\boldsymbol{y}_j$ is given by

$$\frac{b}{(\epsilon + \|\boldsymbol{y}_i - \boldsymbol{y}_j\|^2)(1 + \|\boldsymbol{y}_i - \boldsymbol{y}_j\|^2)}(1 - w(\boldsymbol{x}_i, \boldsymbol{x}_j))(\boldsymbol{y}_i - \boldsymbol{y}_j),$$

where ϵ is also a hyperparameter placed to prevent division by 0 and is usually very small.

The complexity of the first step of UMAP (constructing the temporal graph $\overline{G}$) is $O(N^2H + N^2k)$ if we find the k nearest neighbors by iterating over all the dataset, compute each distance doing $O(H)$ work and then loop over the distance matrix k times to find the k nearest neighbors (assuming that we find the σ_i real values in constant time). From this graph, it only takes $O(N^2)$ work to construct G (matrix addition and component-wise multiplication). We can't frame how long the optimization process takes, but we do know that it scales with the amount of edges $O(Nk)$.

McInnes et al. [24] have developed an open source implementation of UMAP, which we use in our experiments.

2.4. K-means clustering

Clustering also plays an important role on our approach. In this section, we will explain the K-means clustering algorithm, which is the one we used to determine the clusters after performing the multiple dimensionality reduction techniques. In this section we will follow the description of the algorithm given in [1]. Suppose that we have used any of the techniques described above to get an embedding $\{\boldsymbol{z}_n\}_{n=1}^N$ in $\mathbb{R}^L$ of the original $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, for $L < H$.

In K-means clustering, we fix a positive integer K and we consider K vectors $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K \in \mathbb{R}^L$, placed randomly (or by sampling $\{\boldsymbol{z}_n\}_{n=1}^N$). Each of these will represent the prototypical member of cluster k, and we will try to place these $\{\boldsymbol{\mu}_k\}$ vectors as centers of the K clusters.

The process by which we place this cluster centers is via an optimization problem. We can phrase this optimization problem after introducing the indicator variables r_{nk}, given by 1 if $\boldsymbol{z}_n$ belongs to cluster k and 0 otherwise. Using these, we can consider the following objective function:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|^2. \tag{2-14}$$

That is, the sum of squared distances from each cluster member to its center $\boldsymbol{\mu}_k$. Minimizing this function doesn't look as trivial as taking gradients, since we are not only finding the centers $\{\boldsymbol{\mu}_k\}$, but also the indicator variables r_{nk} (which are binary and depend on the position of $\boldsymbol{\mu}_k$). One way of minimizing this function is with an iterative two-step procedure in which we first compute the binary indicators r_{nk}s according to the actual position of the

centers and secondly we optimize J with respect to the $\boldsymbol{\mu}_k$s, leaving the indicator variables fixed.

Computing the r_{nk} binary variables if we fix the centers is trivial: for a given $\boldsymbol{z}_n$, we find the $\boldsymbol{\mu}_k$ that minimizes the distance between $\boldsymbol{z}_n$ and it. In that case, we set $r_{nk} = 1$ and $r_{nj} = 0$ for $j \neq k$. Optimizing (2-14) with respect to the centers involves taking the gradient of J with respect to $\boldsymbol{\mu}_k$,

$$\begin{aligned}\nabla_{\boldsymbol{\mu}_k} J &= \sum_n r_{nk} \nabla_{\boldsymbol{\mu}_k} \left(\|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|^2 \right) \\ &= \sum_n r_{nk} (\boldsymbol{\mu}_k - \boldsymbol{x}_n),\end{aligned}$$

and after setting this gradient to 0 we get that the critical points for the $\boldsymbol{\mu}_k$'s are given by

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^N r_{nk} \boldsymbol{z}_n}{\sum_{n=1}^N r_{nk}},$$

this explains the name of the approach and the notation of $\boldsymbol{\mu}$, since the minimizers of the objective function are precisely the means in each cluster. We are indeed finding K means, and using them as seeds of a clustering.

Using these formulas, we can write the complete pseudocode:

1. Start by placing K vectors $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K$ randomly (or by selecting K of the $\{\boldsymbol{z}_n\}_{n=1}^N$) in $\mathbb{R}^D$.
2. Repeat until convergence:
   a) Compute the indicator variables r_{nk}, given by the equation

$$r_{nk} = \begin{cases} 1 & \text{if } k = \operatorname{argmin}_{k' \in \{1,\ldots,K\}} \left(\|\boldsymbol{z}_n - \boldsymbol{\mu}_{k'}\| \right), \\ 0 & \text{otherwise.} \end{cases} \tag{2-15}$$

   b) With these r_{nk} indicators, update the centers to

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^N r_{nk} \boldsymbol{z}_n}{\sum_{n=1}^N r_{nk}}$$

   for $k \in \{1, \ldots, K\}$.

Convergence follows from the fact that, after each iteration, we are reducing the value of the objective function (2-14). It could, though, converge to a local minima. The algorithm presented here is slow, but variants have been implemented using tree data structures [4].

We used scikit-learn's implementation of K-means clustering.

2.5. Summary

In this chapter we presented 3 different Dimension Reduction techniques to embed points in $\mathbb{R}^H$ into $\mathbb{R}^L$, with $L < H$:

- PCA, which projects the data into the L first eigenvectors of the covariance matrix sorted by eigenvalues. This results in an embedding to the linear submanifold that maximizes the amount of variance (or spread) of the data;
- Isomap, which approximates global geodesic distances in the manifold by constructing a K-nearest neighbors graphs, computing a distance matrix with an all-pairs shortest paths algorithm and then using Multidimensional Scaling to embed it into $\mathbb{R}^L$, and finally
- UMAP, which computes a topological representation of the high dimensional data and embeds it by minimizing a function that measures the difference between a given embedding's topological representation and the high dimensional topological representation. Contrasted with Isomap, UMAP focuses on locally modeling the data, instead of approximating the global distance structure of the manifold.

Finally, we reviewed the K-means clustering algorithm, which iteratively places K mean vectors by minimizing each cluster's distance to its mean. Each iteration of this optimization consists on two steps: recalculating which vectors belong to which clusters, and then moving the means in order to minimize the distance of all cluster points to their respective cluster.

3. Behavioral Repertoires Imitation Learning (BRIL)

3.1. Motivation

Consider the problem of strategic decision making: an agent finds itself in an environment, knowing a list A of actions it can take in order to change the current state s of the environment. Suppose that each state carries with it a measure of reward $r(s) \in \mathbb{R}$, and that we task the agent to maximize this reward signal. How should the agent go around trying the multiple available actions $a \in A$ for each state?

The beauty of framing the strategic decision making problem this way is that the language is so general, that it allows us to describe multiple circumstances using the same concepts. In robot control, for example, the agent is a robot navigating an environment it can sense through its sensors. A Go board has approximately 10^{180} possible states [27], and a player must choose the best possible action to transform the state into one in which she's more likely to win. A group of researchers deciding on the best medicine for an illness in a randomized trial have a set of medicines, each of which results in a response from the patients.

This is the central problem of Reinforcement Learning [35] and, in order to solve it, we are essentially tasked with finding an optimal policy π that maps states s to probability distributions over actions $a \in A$ (see definition 1.6.1).

Several approaches to learning an optimal policy have been researched, varying from tree search methods and Dynamic Programming to the use of Artificial Neural Networks in Deep Reinforcement Learning. By using these different approaches, we have achieved super-human level on board games such as Chess, Go and Shogi [32, 33], and super-human level on videogames such as classic Atari games [25], DoTA 2 [28] and recently StarCraft 2 [37].

Even though these results are incredible, the techniques used do not usually generalize well to changes in the environment. Models that are trained with these techniques tend to overfit and exploit a particular component of the environment-state-action relationship [18]. This implies that there is intrinsic value in trying to learn multiple behaviors in order to properly adjust to changes in the environment.

There has been recent developments in this direction: in [5], the authors pre-compute through simulations a *map of behaviors* for a quadrupedal robot using the illumination algorithm

MAP-Elites [26], with the aim of optimizing walking speed. Upon deployment, they drastically change the environment by damaging the robot on purpose. The authors implement the **Intelligent Trial and Error** algorithm, in which the robot searches for new behaviors in the map that was previously computed for a new, better suited behavior. In this process, the robot updates the map it had precomputed to reflect the fitness of all behaviors in this changed environment, and models his confidence using Gaussian Processes.

In [17], the authors solve the problem of learning a policy for high-level strategic decision making in StarCraft using Feed-Forward Neural Networks. They start by gathering a set of demonstrations from human players, creating state-action pairs and then training a neural network to solve the problem of predicting the next action given a state. After that, they use this model in a simple bot and compare how it fares against the built-in AIs.

This approach suffers from problems analogous to the ones described for Reinforcement Learning techniques: it only learns a policy that expresses the "average" behavior in the dataset of demonstrations. In order to solve this problem, we proposed and used the **Behavioral Repertoires Imitation Learning** (BRIL) approach in StarCraft II [16].

In this chapter we describe BRIL. In the rest of this thesis, we do further research on the relationship between the unsupervised technique used for the embedding step of this approach and the ability of the resulting policy to express multiple behaviors. To do so, we experiment with the three Dimension Reduction techniques described in Chapter 2, and analyze how their theoretical properties affect the expressiveness of the trained policies.

3.2. Behavioral Repertoires Imitation Learning

Inspired by the idea of using behavioral maps and features, we proposed the **Behavioral Repertoires Imitation Learning** (BRIL) technique [16]. With BRIL, models that express multiple behaviors can be trained. In this section we present this novel approach. We will start with discussing the definition of a behavior space, and then we will present how to train models that are able to use this behavior space to express multiple behaviors.

A behavior space is encoded as a subset of $\mathbb{R}^n$ (for some $n \in \mathbb{N}$). Each dimension in this space conveys relevant information about the behavior of an agent in an environment. Going back to the examples described in the motivation, The ELO score of Chess and Go players can be considered a behavior space in $\mathbb{R}$, the different amounts of active components in a pill used in the medical trial can constitute a behavior space in some $\mathbb{R}^n$, the amount of time each leg touches the ground in a quadrupedal robot can also form a behavior space in $\mathbb{R}^4$.

Usually, behavior spaces require more than 2 behavioral descriptors in order to accurately portray different behaviors. For this reason, we propose the use of Dimension Reduction algorithms in order to embed these spaces into the plane. This embedding allows us to visualize the data and detect relationships between behaviors. After this reduction, we end up with a set $\mathcal{B} = \{(b_1, b_2)\} \subseteq \mathbb{R}^2$ of **Behavioral Features**. Of course, this dimension

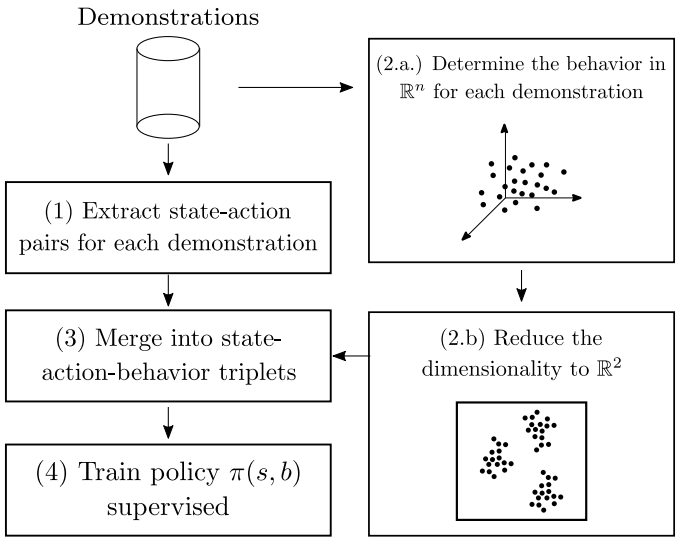

Figure 3-1.: Behavioral Repertoires Imitation Learning. After gathering a dataset of demonstrations, expressive policies $\pi(s, b)$ can be trained by expanding the state-action pairs with behavioral features. These behavioral features are obtained after embedding in low dimensions a high dimensional behavior space.

reduction can be performed to arbitrary dimensions, and the choice to embed into $\mathbb{R}^2$ is only justified by its convenience for visualization.

In the usual setting, state-action pairs are gathered and they are used in a Supervised Learning algorithm to learn a policy $\pi(s)$ that maps states to probability distributions over actions. In BRIL, we first determine a low dimensional behavior space $\mathcal{B} \subseteq \mathbb{R}^2$, and we use these behavioral features to augment the input of the Supervised Learning technique. As a consequence, we actually learn a joint distribution $\pi(s, b)$ on both states s and behaviors $b = (b_1, b_2)$.

In deployment, we can use this distribution as policy by conditioning on a particular behavior. If we fix $b \in \mathcal{B}$, we can use $\pi(\cdot, b)$ to decide on actions to take given states s. This new policy will express the behavior b.

We can summarize the BRIL approach with the following steps:

1. A dataset of demonstrations is gathered.
2. State-action pairs are extracted from this dataset.
3. A behavioral space is designed using this dataset and expert knowledge.
4. A low dimension embedding and clustering of this behavioral space is computed in order to form a two-dimensional map of possible behaviors.
5. The coordinates of this embedding are used as behavioral features, and are appended to form state-behavior-action triplets.
6. A policy is learned using these state-behavior-action triplets, where the inputs are the

states and the behaviors.

This process is summarized in Figure **3-1**. Since the result of BRIL are policies that can be conditioned on behaviors, this method can be applied in order to kickstart more general Reinforcement Learning methods that deal with tournament-like structures in which the initial agents are gathered from Imitation Learning (just like the AlphaStar League in [37]), as well as for training agents and conditioning them to avoid catastrophic behaviors that could be found in the dataset.

3.3. States, actions and behaviors in StarCraft 2

In the next chapter, we discuss the experiments designed to measure the impact of the dimension reduction algorithm in BRIL. We use, as discussed in previous chapters, the video game StarCraft 2. In the spirit of giving context, this section will explain how states, actions and behaviors are usually encoded by the community when building bots. Most of this discussion is summarized from [27].

The community tackles the problem of creating human-level bots by dividing the multiple tasks an agent confronts into smaller, more focused problems, implementing different managers to solve them. Most bots in previous competitions have for example a worker manager, which is in charge of optimally distributing worker units; an assault manager, which decides when to attack or defend and where; a production manager, which tackles high-level decision making by reading the current state and deciding which action to take.

3.3.1. States

States in StarCraft can be encoded in multiple ways. First, we could encode them as we do in chess or Go: considering all possible combinations of unit types and positions. This upper bound is of order 10^{1685} (for more details, follow [27]). Even though this encoding is helpful in tree-search methods in these classical games [33], the nature of the game invites us to consider a different way of encoding states, due to the fact that it is real-time and not turn based.

The community considers different *state abstractions*. The worker manager would then work at a *micro* level in this state abstraction, handling individual units and focusing on their position; the production manager, on the other hand, would work on a *macro* level, deciding upon high-level descriptions of the state: amount of resources and units, enemy units scouted and so on. In our experiments, we tackle the problem of *macro* decision making. The complete description of our state abstraction can be found in Sec. 4.1 of Chapter 4.

3.3.2. Actions

Just like there exist multiple levels of abstraction for states, actions are defined per module, that is, per level of abstraction. Actions of *micro* level abstraction are e.g. *move this unit to this location*, *attack this enemy unit with this allied unit* and so on, while actions on a *macro* level are *train this type of unit*, *build this type of building* or *research this type of upgrade*. As stated above, our experiments focus on high-level decision making. There are 70 possible actions in all our experiments, which are those involving the creation of units and the research of upgrades.[1]

3.3.3. Behaviors

When playing StarCraft 2, agents are able to express behaviors like being aggressive and quick or playing defensive. These behaviors relate mostly to their **unit compositions** and **build orders**.

Their unit compositions relates mostly to what their armies are composed of. In the game, the player can find low-cost units that allow for quick and inexpensive attacks, or relatively expensive units that allow for powerful, yet slower, attacks. In the case of our experiments, which involve the Terran race, there are two families of unit compositions: those which are related to *biological units* (or *bio*, for short) and those related to *mechanical* units (or *mech*).

Build orders, on the other hand, encode the order in which buildings and units were built or trained. The game's community shares these different build orders and discusses them thoroughly in public forums, and tools that analyze different build orders automatically have been developed.[2] Recent research has focused on the problem of clustering the different build orders present in a family of replays [8].

In this thesis we tackle behavior spaces that encode allied unit composition. Experiments with build order encodings were also performed, but the results were unsatisfactory and require future work.

[1] It is worth noting that recent developments in RTS AI have showed that an *end-to-end* model (which controls mouse and keyboard and has no particular abstraction but the information coming from the game) is able to beat professional level humans. See [37] for more details.

[2] https://lotv.spawningtool.com/

4. Experiments and results

In order to investigate the impact of the dimensionality reduction process in the behavior adaptation of the novel approach presented in Chapter 3, we used the video game StarCraft II. In Real-Time Strategy games, players are able to express plenty of strategies and behaviors [2]. This fact makes StarCraft II an excellent testbed for our approach.

In this chapter we describe the experiments that were performed using the video game, and we discuss their results. In total, we ran the following 5 experiments:

1. We trained a neural network model that serves as a baseline, replicating the results of [17] in StarCraft II.
2. We designed a high dimensional behavior space and embedded it into the real plane using PCA, Isomap and UMAP. We also used K-Means to clusterize these low dimensional spaces. We also embedded it into 8 dimensions using PCA in order to explain more variance in the data.
3. We used the low dimensional behavioral features that resulted from these embeddings to train 4 neural network models, with the aim of being able to modulate their behavior by specifying points on the respective map.
4. We tested all 5 models (the baseline and the 4 embedding-related models) by using them as the production managers[1] of a simple open source bot called `sc2bot`[2], developed by Niels Justesen. Everything besides the production manager is hard-coded using simple heuristics for middle and micro level unit control.
5. We learned the win probability surface against a fixed opponent (the built-in AI in *easy* difficulty setting) by fitting a Gaussian Process Classification to each 2-dimensional embedding after sampling 100 points from it and running them in the bot, getting either a 0 for a loss or a 1 for a win.

This chapter starts with a small description on the dataset generation process, and then continues with a detailed description of each experiment and their respective results. Detailed information about the computational tools used for gathering the data, training the models, the bot's implementation and the Gaussian Process Classification can be found in Appendix A.

[1] A production manager is the module that is in charge of deciding which units to to train/build and which upgrades to research. For more context on the structure of bots, follow Sec. 3.3 of Chapter 3.

[2] https://github.com/njustesen/sc2bot

The experiments that were carried out in this thesis are similar to those we performed in [16]. Nevertheless it must be noted that, since the behavior spaces are different, these results are novel and extend the study about the aptitudes and defects of our approach. Moreover, these results focus on the characteristics that the embedding process must have in order for the approach to work, and expands the experiments of the article by using a continuous approximation of the win probability per space instead of a discrete K-armed-Bandit problem.

4.1. Gathering the data

As we saw in Chapter 1, a dataset $\mathcal{D} = \{(\boldsymbol{x}_n, a_n)\}_{n=1}^{N}$ of state-action pairs must be gathered in order to train models to approximate policies. For this purpose, we developed an open-source tool called `sc2reaper`[3] to consolidate the data of 7777 replay files of Terran vs. Zerg provided by Blizzard and DeepMind [38]. In total, 1,625,671 state-action pairs were extracted from these replay files, following the state abstraction specified in [17], which consists of

1. Resource information (amount of minerals and vespene gas).
2. A description of supply (total supply, available supply, supply in workers and army).
3. The amount of allied units of each type.
4. The maximum amount of enemy units of each type scouted.
5. The amount of allied units currently being built.
6. The (highest) progress of the allied units which are currently being built.
7. A binary indicator for each possible allied upgrade.
8. The progress of upgrades currently being researched.

There are 70 possible actions, and these were encoded using 1-hot vectors.

More complete abstractions of the state space in StarCraft have been considered (including, for example, geographic information in the form of a convolutional network input processing the minimap [22]). In our work, though, we focus on setting a leveled comparison between BRIL and the work described in [17]. Thus, we only consider this state abstraction.

4.2. Training a baseline

Using the dataset of state-action pairs described above, we trained a neural network with 3 hidden layers and 256 hidden nodes per layer. This topology was fixed after a grid search of best test accuracy for hidden layers in $\{1, 2, 3, 4\}$ and hidden nodes in $\{64, 256, 512\}$. After 10 successive training experiments, the models performed in average with a 47.1% test

[3] https://github.com/miguelgondu/sc2reaper

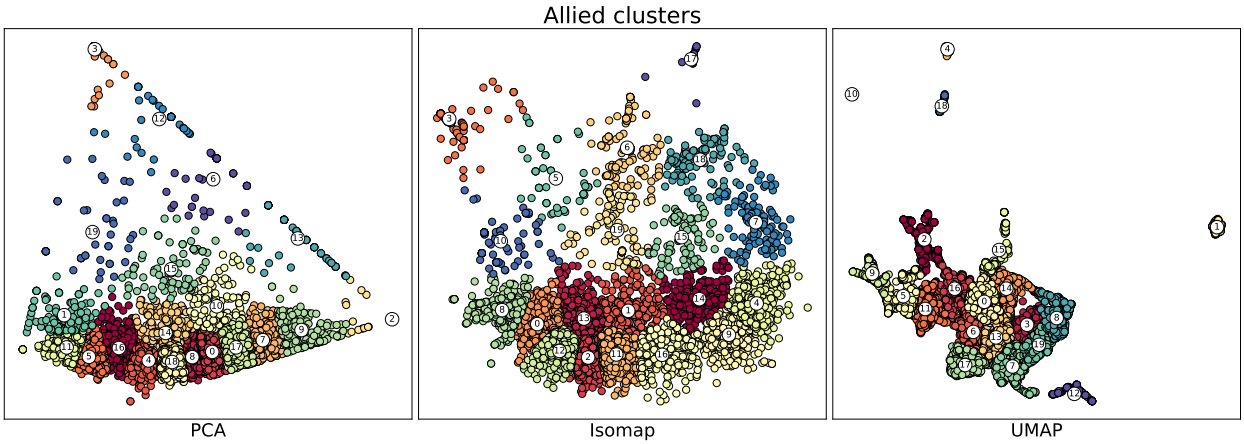

Figure 4-1.: Clustering of the Allied Units behavioral space.

accuracy ($2\sigma = 0.2\%$). This accuracy is similar to previous results achieved by Justesen and Risi [17] (46.4% test accuracy). The test loss for this baseline model is presented in Table 4-1.

Even though a test accuracy of 47.1% might seem small, notice that the problem of predicting the next human action given a dataset of state-action pairs does not constitute an exact classification problem, in the sense that a single state might lead to different actions. Since this relation is non-functional in the strict sense (that is, it is one-to-many), Feed-Forward Neural Networks can only approximate it. This can be considered a motivator for the BRIL approach we are testing: learning more expressive policies allows us to model this one-to-many behavior of states and actions by introducing the behavioral features.

4.3. Designing a low-dimensional behavior space

Our second experiment consisted on designing a behavior space and using Dimension Reduction techniques to get a low dimensional representation of them. For a summary of the 3 techniques we used, see Sec. 2.5.

Using the information extracted and processed from the 7777 replays, we designed the **allied unit behavior space** as follows: there are 20 different types of allied military units, which we will list with an index $i \in \{1, \dots, 20\}$. For each replay we compute a point $\boldsymbol{u} = (u_i)_{i=1}^{20} \in \mathbb{R}^{20}$ in which every coordinate u_i is the maximum amount of unit i seen in the replay. We later normalize these points into $\hat{\boldsymbol{u}} = \boldsymbol{u}/\|\boldsymbol{u}\|_1$, where $\|\cdot\|_1$ is the 1-norm of $\mathbb{R}^{20}$, in order to get a vector of percentages. These points $\hat{\boldsymbol{u}}$ constitute the high dimensional behavior space.

4.3.1. Embeddings

This behavior space was then embedded into the plane using the three different dimensionality reduction algorithms described on Chapter 2: PCA, Isomap and UMAP. After the embedding, K-means clustering was used to group the behaviors into 20 clusters. Figure **4-1** shows the result of this experiment. Embedding with PCA in two dimensions explained 47.65% of the variance in the original set of points (see Eq. (2-6)).

An additional embedding was computed by using PCA to project to the 8-dimensional space. This number of coordinates was chosen for PCA to explain 89.6% of the variance in the data. We will refer to this model as PCA8 in order to distinguish it from the 2-dimensional embedding made with PCA. This embedding, unfortunately, can't be properly visualized in 2 or 3 dimensions.

The embeddings presented on figure **4-1** show interesting structures, and in the following paragraphs we will analyze each embedding with respect to the theoretical properties and foundation of each method.

The PCA embedding shows a projection into a quadrilateral shape, in which 3 of the corners (clusters C2, C3, C11) are distinctively far apart. A good explanation for why PCA projects the data to this quadrilateral might be the fact that we are essentially projecting a 20-dimensional unit hyper-tetrahedron[4] bounded by the unit cube $[0, 1]^{20}$ to a plane (i.e. the span of two vectors). The vertices of this quadrilateral are then to be considered extremal strategies, in which possibly one of the coordinates was almost 1 and the other were almost 0.

Theoretically speaking, the Isomap embedding is trying to approximate the global distance structure. Thus, globally speaking, there are a couple of strategies that are separated from the rest (namely, strategies in clusters C3 and C17). We can then hypothesize that these strategies involved units that weren't as present in other strategies, thus increasing the distance of their vectors in the high dimensional space.

Finally, UMAP is able to cleanly separate some clusters from a body of replays in the center. The effort to preserve local structure in UMAP, and the attractive and repulsive forces in its embedding process, is stating that clusters C1, C4, C10 and C18 are to be considered radically different from the ones in the replay mass. Nevertheless, this replay body also has distinctive edges and corners in clusters C2, C7, C9, C12 and C15.

Since PCA8 doesn't allow for geometrical interpretations in 2 or 3 dimensional spaces, we will perform the analysis of its behavior only when deployed in the bot. This will be studied in Sec. 4.5.

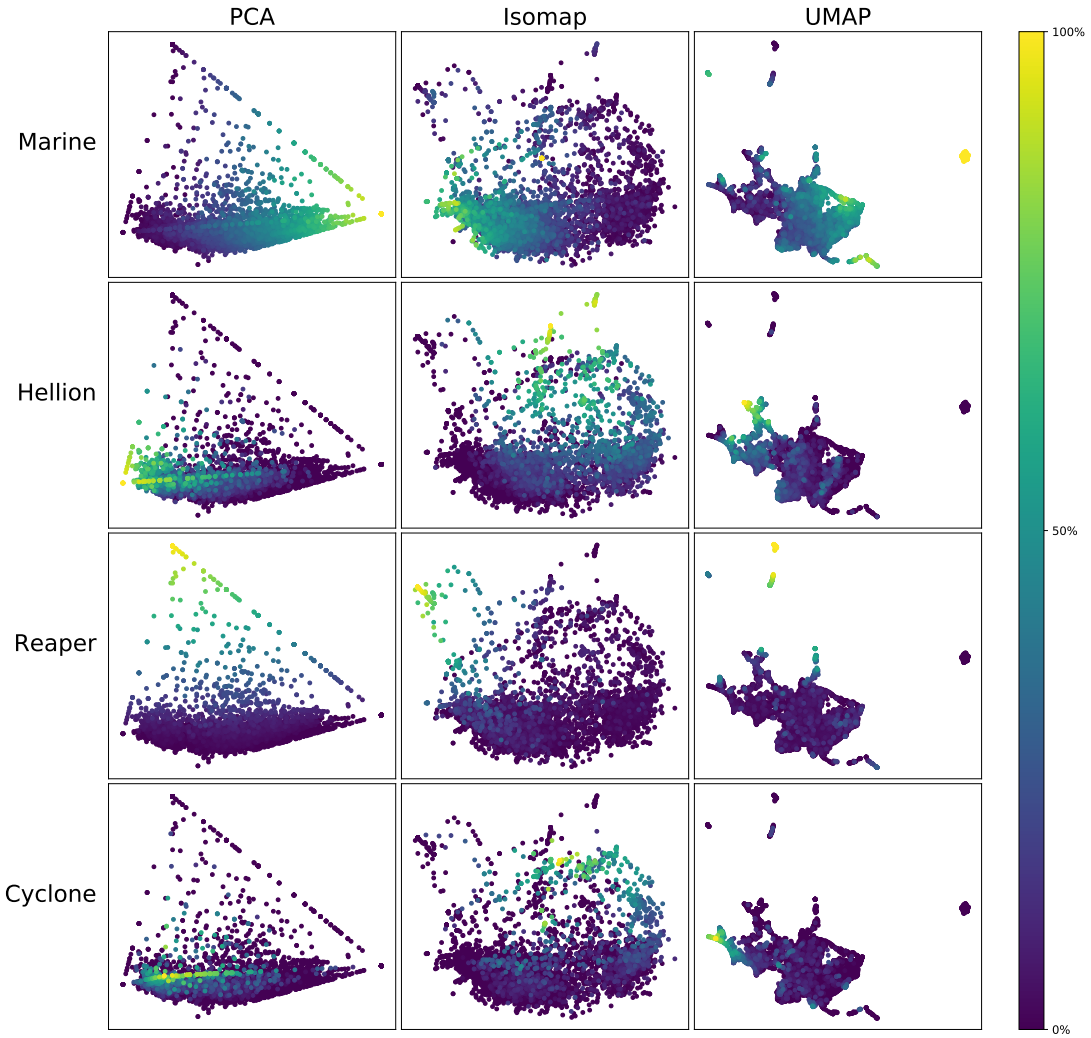

Figure 4-2.: Illuminating the space of allied unit behaviors with the amount of Marines, Cyclones, Reapers and Hellions produced in each game. The colors indicate the percentage of that unit seen, that is, the component of the normalized vector that corresponds to that unit.

4.3.2. Illuminations

In order to study the behaviors that appear in these clusters, we propose the following heuristic: illuminating each point in the 2 dimensional map by how many units of a certain type were built. That is, for each point we illuminate by a particular component of the normalized vector $\hat{\boldsymbol{u}}_i$ in the original high dimensional space. Figure 4-2 shows an illumination of all clusters of the allied unit behavior space by the following units:

- **Marines**, which are the most basic and cheap units the Terran race has. They are involved in most *bio* strategies in high numbers. Marines can also be used in a *rush* strategy, in which the player constructed his army-building facilities near the opponent and attacked as early as possible with a small army of only Marines.
- **Hellions**, fast car-like units with the potential of quick run-by attacks, yet very fragile and only good against light-weight units. Hellions can be transformed into Hellbats, a more robust but slow unit.
- **Cyclones**, mechanical units with armor piercing capabilities. These could be considered the mid-tier units for *mech* strategies.
- **Reapers**, a unit designed for early scouting and harrasment. These are also the subject of *rush* strategies due to their mobility and sustain.

Figure 0-1 shows a screenshot of the game in which Marines, Hellions and Cyclones are showed.

These illuminations confirm the hypotheses that were raised in the last subsection about the strategies that were being distinguished by the different embeddings:

In PCA, the distinguished vertices of the quadrilateral correspond to strategies involving armies composed mostly of Marines (C2), Reapers (C3) and Hellions (C11), and a gradient can be seen in which the amount of e.g. Marines decreases gradually from right to left. Such gradients are to be expected from the method, since all points in the Marine axis are getting projected to a plane, where we expect the 1 to be at cluster C2 and the 0 to be somewhere between cluster C3 and cluster C11. The same phenomena happens for the 3 different kinds of units. Notice, however, that the gradient of Reapers going downwards is rather small. This is to be expected, since Reapers are units that do not usually appear in higher numbers in other strategies besides *Reaper Rush*, an early attack strategy in which only Reapers are used. The fact that precisely Marines, Reapers and Hellions are in the vertices of this quadrilateral can be explained by their nature in the game: these 3 units, when employed, are used in excessive numbers. This implies that their vector representation in high dimensions is close to being the unit vector in a given direction, and PCA is finding the most variance by considering a plane parallel to the one generated by these 3 army compositions in $\mathbb{R}^{20}$.

[4] By this, we mean the set of points $\boldsymbol{x} \in [0,1]^{20}$ such that $\sum_i x_i = 1$, where $\boldsymbol{x} = (x_1, \ldots, x_{20})$.

Method	Mean Test Accuracy ($\pm 2\sigma$)	Mean Test Loss ($\pm 2\sigma$)
Baseline	$47.10 \pm 1.97 \times 10^{-1}$	$6.08 \times 10^{-3} \pm 4.07 \times 10^{-4}$
PCA	$47.22 \pm 2.13 \times 10^{-1}$	$6.10 \times 10^{-3} \pm 3.63 \times 10^{-4}$
Isomap	$47.17 \pm 1.70 \times 10^{-1}$	$6.04 \times 10^{-3} \pm 2.81 \times 10^{-4}$
UMAP	$47.20 \pm 1.49 \times 10^{-1}$	$6.11 \times 10^{-3} \pm 4.07 \times 10^{-4}$
PCA8	$47.27 \pm 1.68 \times 10^{-1}$	$5.93 \times 10^{-3} \pm 3.09 \times 10^{-4}$

Table 4-1.: Mean Test Accuracy and Error. This table shows the Mean Accuracy and Mean Error on the test set for the Baseline and the models that were augmented by the features of the allied behavior space. The Baseline performed on par with previous research done on the problem of predicting actions from states, and the differences between the results of the baseline and the results of the augmented models imply that, by augmenting with behavioral features, the model doesn't perform better in the predicting task.

Cluster C3 in Isomap also corresponds to a *Reaper Rush*, and the fact that this strategy is separated (in terms of global distance) from the other replays can also be explained by the fact that Reapers are not frequently used in high numbers in other strategies. The same phenomenon seems to be happening with cluster C17, which in this case corresponds to a strategy in which mostly Hellions are involved. Since Marines are the basic unit of Terran and they appear in plenty of strategies, the clusters involving Marines are subsets of the main body of replays, because, in terms of the approximated geodesic distance, they are close to most strategies.

UMAP's embedding is showing that the strategy of pure *Marine Rush*, a strategy in which the army composition consists of only Marines, is separated from the rest in cluster C1. UMAP also separates the pure *Reaper Rush* strategy in cluster C4. The fact that, unlike PCA or Isomap, UMAP completely separates the *Marine Rush* strategy from strategies that involve Marines in a lesser degree is worth analyzing. This might be the case because UMAP takes into consideration the local structure, and in this case, the local structure of these rush strategies induces great repulsive forces. Another showcase of UMAP's local understanding of the behaviors is the fact that, unlike PCA and Isomap, this method separates the Hellion-oriented strategies from the Cyclone-oriented strategies in clusters C2 and C9 respectively.

4.4. Training networks with behaviors

After computing these low-dimensional representations of the behavior space, we added the behavioral features as inputs to the dataset and we divided it into 3 parts for training (60%), validation (30%) and test (10%) per cluster. We trained a total of 10 models for each of the

Model	Cluster	Wins/Total	Amount of Unit created on average (σ) Marine	Cyclone	Reaper	Hellion
Baseline	N/A	51/100	30.73 (28.25)	1.41 (4.91)	0.44 (0.74)	0.21 (0.65)
PCA	C2 (Marines)	64/100	**39.77 (29.05)**	0.26 (1.15)	0.19 (0.48)	0.02 (0.14)
	C5 (Cyclones)	9/100	1.88 (3.08)	2.12 (3.12)	0.43 (0.85)	0.93 (1.78)
	C3 (Reapers)	24/100	12.49 (17.23)	**2.7 (3.83)**	**1.42 (1.52)**	0.22 (0.83)
	C11 (Hellions)	13/100	1.51 (4.34)	2.37 (3.38)	0.44 (0.94)	**1.67 (4.39)**
Isomap	C8 (Marines)	39/100	24.4 (19.69)	0.34 (0.94)	0.61 (0.77)	0.09 (0.45)
	C18 (Cyclones)	42/100	13.08 (14.99)	2.08 (4.12)	0.32 (0.51)	**2.79 (4.47)**
	C3 (Reapers)	75/100	**44.67 (30.14)**	1.04 (2.53)	**0.74 (0.90)**	0.29 (1.13)
	C6 (Hellions)	48/100	19.51 (18.27)	**2.38 (4.10)**	0.39 (0.63)	2.65 (7.39)
UMAP	C1 (Marines)	85/100	**52.95 (26.03)**	0 (0.00)	0.04 (0.20)	0.22 (1.14)
	C9 (Cyclones)	49.5/100	15.29 (15.05)	**0.6 (1.19)**	0.21 (0.52)	**7.28 (10.76)**
	C4 (Reapers)	73/100	35.48 (22.15)	0.09 (0.55)	**1.11 (2.93)**	3.17 (7.78)
	C2 (Hellions)	66.5/100	24.66 (17.94)	0.39 (1.04)	0.33 (0.75)	6.18 (10.30)
PCA8	C5 (Marines)	77/100	**53.77 (28.25)**	0 (0.00)	0.02 (0.14)	0.03 (0.30)
	C10 (Cyclones)	50/100	11.1 (15.69)	**3.55 (3.25)**	0.15 (0.36)	7.93 (11.24)
	C4 (Reapers)	14/100	5.53 (11.78)	0.64 (1.49)	**1.04 (1.27)**	0.19 (1.06)
	C7 (Hellions)	45/100	8.75 (10.67)	0.48 (0.90)	0.12 (0.32)	**15.71 (14.50)**

Table 4-2.: Average Units trained by each model after 100 games. This table shows the amount of units that each model built on average after being fed the cluster centers of 4 different clusters. These results show that the different models do modify their behavior when given different behavioral features.

4 embeddings, varying the seed in the stochastic gradient descent algorithm. The mean test accuracy and mean test loss are presented in Table 4-1.

These results indicate that the network isn't extracting any new information from the features at prediction time since, after adding the behavioral features, there is no significant increase in test accuracy. This result is interesting since, in some sense, we are giving the network a summarized, low dimensional representation of the actual units that were built in each replay. The network isn't inferring the original high dimensional vectors from these behavioral features since, if it did, it would have access to the actual percentages of units that were built.

4.5. Deploying models in-game

Once we had models trained on the inputs of the dataset and the behavioral features of each embedding (arriving at multiple policies of the form $\pi(s, b)$), we tested our ability to modify their behavior by using 4 different cluster centers $b_{c_{i_1}}, \ldots, b_{c_{i_4}}$ as input in the policies. These behavior centers were selected to represent the clusters that were studied during our analysis of the illuminations (i.e. Marine, Cyclone, Reaper and Hellion).

We chose one of the 10 models trained for each embedding at random, and we tested it on each of the 4 clusters described above for 100 games against a fixed opponent (the built-in Zerg AI in *easy* difficulty) using `sc2bot`. We also ran 100 games with one of the models trained for the baseline. Since the output of the models is a probability vector over the actions, the bot decides which action to take by sampling this discrete probability, instead of focusing on greedy approach (in which only the action with the most probability according to the output is taken). This introduces noise into the measurements, but allows the bot to perform better.

The units built by each of these models in average are presented in table **4-2**. These results show that the behavior of the bot can be manipulated by using different coordinates in each embedding, as we will now discuss.

When using the PCA embedding's behavioral features, the model focuses on training Marines when using cluster C2's center, and then shifts to training significantly less Marines when instructed to follow the behaviors of clusters C5 and C11. This suggests that the model is able to replicate the strategy of replays in cluster C2. This same phenomena happens at clusters C3 (Reaper oriented strategies) and C11 (behaviors with Hellions). The model, thus, is notably changing its behavior when different behavioral features are used as inputs.

Similar behavior changes are also expressed in Isomap. In this case, however, the model is not exactly expressing the behaviors it is intended to replicate. For example, it builds the most amount of Marines when following Reaper oriented strategies. Nevertheless, it continues to be the case that the behavior of the agent is being influenced by the pair of behavioral features being used.

UMAP's separation of the *Marine Rush* strategy is arguably passed on to the model's behavior when using cluster C1's center. At these coordinates, the model focuses on training Marines to the extent of, for example, not training any Cyclones and training almost no Reapers. Another interesting property of the model trained on UMAP's embedding is that it is training significantly more Marines in all strategies in comparison to PCA. This model is also building more Hellions on average that all other models.

Finally, PCA8 exhibits the cleanest separation of behaviors. After running 20 games in each cluster, we identified the ones that possibly exhibited the corresponding behaviors (focusing on Marines, Cyclones, Reapers and Hellions). Once these clusters were identified, 80 more games were played in order to compare it to the other methods. PCA8 presents two interesting properties: it is able to separate the *Marine Rush* strategy just like UMAP, and it also manages to separate the other clusters from Marine-related strategies, just like in PCA (in two dimensions). Moreover, we could arguably say that, when running the policy on these cluster centers, their respective behavior was replicated. This fact implies that accounting for more variance in the data is correlated with cleaner separation and better expressivity of the different behaviors. Nevertheless,accounting for more variance by projecting into $\mathbb{R}^8$ instead of $\mathbb{R}^2$ comes with the drawback of having to manually identify each behavior by running

games in all clusters, instead of being able to rely on visualizations and illuminations, like the ones discussed for the other 3 embeddings.

It is worth framing this discussion in terms of the theoretical properties each Unsupervised technique has, and analyze the behaviors of the models accordingly.

The linear embedding of PCA leads us to the belief that distance is playing a major role in the policy each model is learning. Indeed, the selected clusters C2, C3, C11 are as far away as is possible in the quadrilateral, and the illuminations show that e.g. cluster C11 has almost no bearing with Marine related strategies. The same seems to be happening for each pair of opposite corners involving these clusters. The Isomap embedding also reinforces this idea, since clusters C8 (Marines) and C3 (Reapers) are not as separated as in PCA's, and this translates in the behavior of the model, when fed the coordinates of C3, being Marine heavy.

Another interesting phenomena that seems to be happening in all 2-dimensional embeddings is the fact that the x axis is becoming a proxy for determining whether or not to build Marines, since the gradient of building Marines is moving mostly horizontally. This is particularly the case in UMAP's embedding, where close values to 1 indicate a maximum amount of marines.

Finally, the fact that more variance is being explained by PCA8 may induce a cleaner separation of the 4 discussed strategies. Following the analogy of projecting the unit hypercube in $\mathbb{R}^{20}$, the strategies might get separated as vertices in the 8-dimensional linear subspace, thus resulting in a better expression of each strategy.

4.6. Learning the win probability in each map

An advantage of having a policy that expresses multiple behaviors is that we can search for the optimal strategy against a particular opponent. This last experiment goes in this direction. For each embedding, we computed the win probability surface in order to detect the strategies that had the highest chance of winning.

We estimated this win probability surface over each embedding by solving a classification problem using Gaussian Process Classification. This can be considered a binary classification problem, in which the targets belong to $\{0, 1\}$ (either a loss or a win). Non-parametric models are ideal for these types of classifications, since data is scarce and expensive to gather.

We trained a Gaussian Process Classifier with constant mean and Radial Basis kernel (with lengthscale 1). We sampled a set of 100 points of each map, 20 cluster centers and 80 randomly selected points, and ran a game using these points as behavioral features in the model for each embedding. These embeddings and win probabilities can be found in Figure 4-3.

All win probabilities indicate that there is a positive correlation between the number of

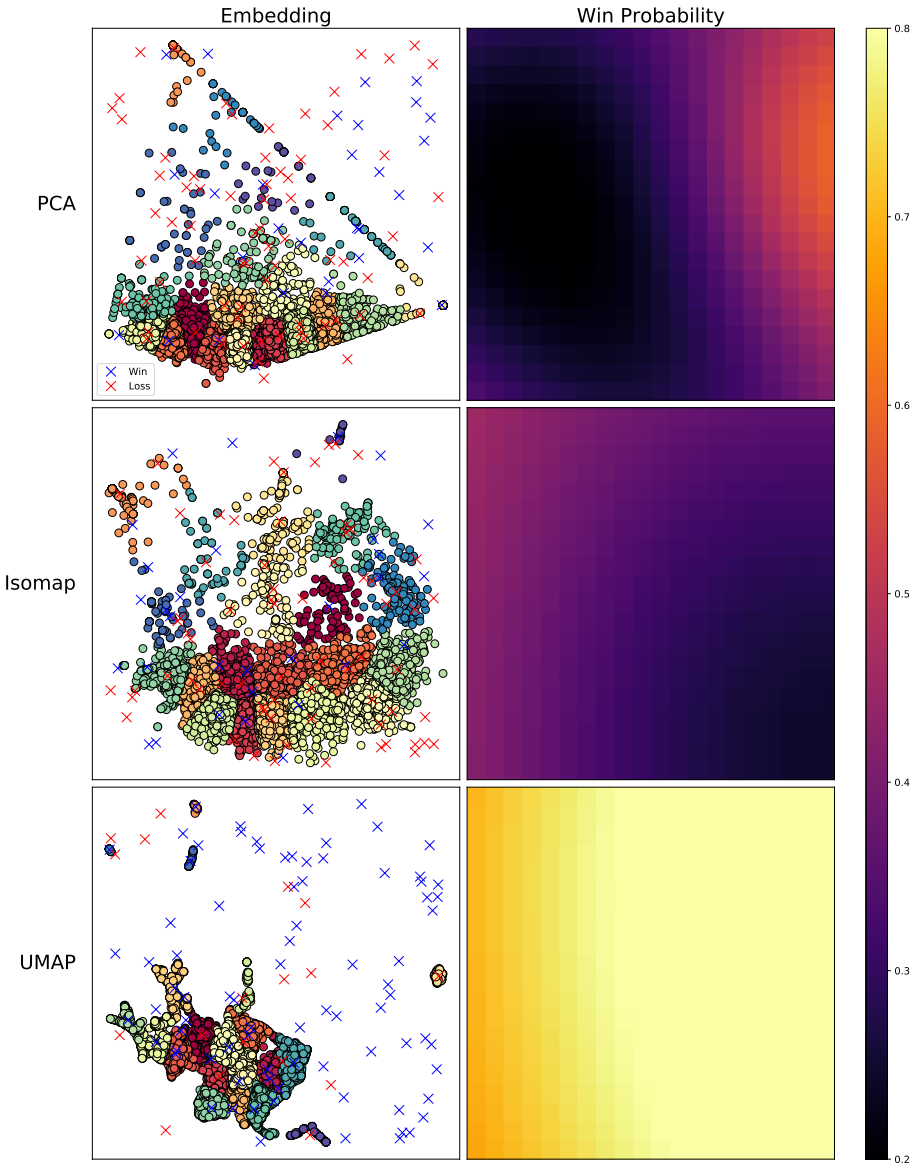

Figure 4-3.: Embedding and Win Probability. This figure shows the results of fitting a Gaussian Process Classification after running the bot in 100 different points, 20 of which were the cluster centers and 80 were selected at random. The blue crosses represent points in which the result was a win, and the red crosses represent points in which the result was a loss. All three embeddings show that the bot is more successful when executing simpler strategies involving Marines (see Fig. 4-4)

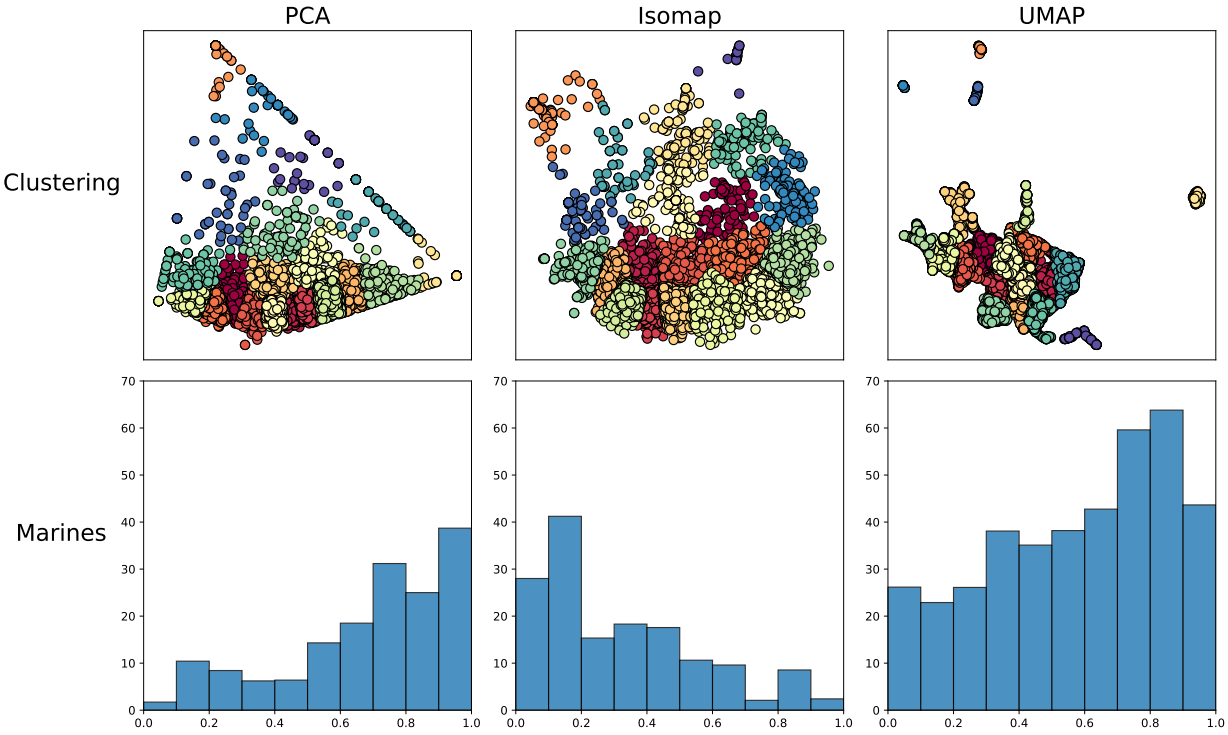

Figure 4-4.: Average Marines trained. We computed the average amount of marines trained in these 100 games for each interval of size $\Delta x = 0.1$. There results show that the network is expressing the behaviors that the original replays had, when comparing it to the Marine Illumination in Fig. 4-2.

Marines trained in the game and the probability of winning (see Fig. 4-4). This can be explained because of the simplicity of the bot, which handles units in a middle and micro level using hard-coded rules about attacking and retreating. Thus, the bot manipulates basic units such as Marines better than more advanced ones like Cyclones.

PCA's win probability surface shows that Hellion and Cyclone related strategies perform worse in average than Marine oriented ones, as is to be expected. An interesting behavior occurs when sampling points from outside the quadrilateral shape, on the upper right quadrant of the unit square. These strategies perform better on average. One reason why might be that the model is learning its context through the x axis, and points to the right of the line $x = 0.5$ are being considered as Marine-based strategies. Another possible explanation, is that the model is identifying different strategies not used by human players that might be more successful for this particular bot against this particular opponent.

In general, the performance of the Isomap model was poor, with a mean win probability of 0.34. Still, the pattern of simple behaviors being successful also occurs, since the win surface is tilted towards the left, favoring Marine-related strategies. In comparison, the UMAP model performed better on average, with a mean win probability of 0.80, and the favoring

of simple strategies can also be seen. We could argue that the fact that the UMAP model is optimal against this opponent is precisely due to how it distributed the Marine-related strategies most clusters (as can be seen in Figure 4-2 and Table 4-2). Thus, the reason why the mean win probability is so high may be because more strategies are using Marines in their compositions.

These results also point us towards thinking that distance is playing a major role in the behavior described by the bot. For example, behaviors that are closer to the Marine cluster in PCA and UMAP exhibit similar responses, even though the training set had no data corresponding to those regions.

4.7. Summary

In this chapter we described the experiments that were performed in order to analyze the relationship between the behavior space, the embedding technique and the expressive policies learned using the BRIL method.

In these experiments we constructed a dataset of demonstrations using our tool, we trained a baseline for comparison, we designed a behavior space, used three different techniques to embed it to lower dimensions and we used these low dimensional behavioral features as inputs in neural network models. We tested these expressive policies using a simple bot, and we used Gaussian Process Classification to learn a win surface on each planar embedding.

Results showed that the low dimensional embeddings embedded strategies at different distances from one another (see Fig. 4-1). This is to be expected, according to the notion of structure that each method is trying to preserve. Table 4-1 suggests that, by augmentating the input with behavioral features, the networks showed no significant improvement in the classification problem in hand.

Nevertheless, these policies were able to replicate the different strategies and behaviors found in the original behavior space. The degree with which these policies replicated the behaviors seems to be related to the distances between the clusters: separated clusters are replicated more precisely (see, for example, the results for UMAP's C1 Cluster in Table 4-2). This suggests that, by trading off visualization for dimension in the embedding process, the resulting clusters could be better separated and, thus, their behavior might get more preciely replicated (see the results for PCA8 in Table 4-2).

Finally, the simplicity of the bot made it so that strategies that use simpler units had a better chance at winning the game, as is studied in Fig. 4-3.

5. Conclusions

In this chapter we present a brief summary of all the thesis, the conclusions that were reached through the research and some comments on future work.

5.1. Summary

This thesis focused on studying the novel approach **Behavioral Repertoires Imitation Learning** (BRIL). BRIL trains more expressive, more adaptive models by augmenting the input of them with **behavioral features**. Thus, instead of learning a policy $\pi(s)$ on just states, these models learn a policy $\pi(s, b)$ on states and behaviors. The way these behavioral features are extracted is by using expert knowledge: an encoding of relevant characteristics is made in an arbitrary Euclidean space $\mathbb{R}^n$ (for some $n \in \mathbb{N}$), and this high dimensional behavior space is projected to lower dimensions using Unsupervised Learning techniques.

We formulated BRIL and tested it in the environment of the video game StarCraft 2 [16]. This first test was very successful, in the sense that we were able to modify the behavior of the agents by selecting different behavioral features b and using them as inputs in the models.

In this thesis, we expand and analyze the approach by using different Dimension Reduction algorithms. We started with introductions to Supervised Learning using Feed-Forward Neural Networks and Unsupervised Learning using Principal Component Analysis (PCA), Complete Isometric Feature Mapping (Isomap) and Uniform Manifold Approximation and Projection (UMAP). After that, we summarized the BRIL approach, and we finished with a series of experiments, designed to inquire about the relationships between the expressiveness of the resulting policies and the embeddings in which they are trained.

The results suggest that the BRIL approach works successfully when it comes to creating models that express different, adjustable behaviors. In the maps in which the methods are trained, distance seems to be playing a vital role in separating behaviors from one another.

5.2. Conclusions

In this section, we enlist this thesis' conclusions:

- We presented an introduction to Supervised Learning using Neural Networks, covering the main definitions and the processes by which these parametric models learn from data.
- A similar, shorter introduction to Gaussian Processes was also reviewed.
- We studied three different Dimension Reduction techniques, with emphasis on the properties and structure they try to preserve from the original high dimensional data:
  - PCA, which maximizes the variance of the data after the projection.
  - Isomap, a technique that approximates the global distance structure by locally joining points with nearest neighbors and computing the all-pairs distances.
  - UMAP, which enforces a uniform distribution of the data and constructs a topological representation by joining locally defined fuzzy simplicial sets.
- The BRIL approach was presented, with the aim of training models that are able to express multiple behaviors.
- In order to test the BRIL approach, we developed an open source tool for data extraction called `sc2reaper`. With this tool, we constructed a dataset of 7777 replay demonstrations of Terran vs. Zerg in StarCraft 2, amounting to a total of 1,625,671 state-action pairs.
- For our experiments, we fixed the topology of the Feed-Forward Neural Network to 3 hidden layers and 256 hidden nodes after a grid search in the sets $\{1, 2, 3, 4\}$ and $\{64, 256, 512\}$ respectively.
- We trained a baseline model, replicating previous results on StarCraft.
- We designed the allied unit behavioral space, in which the components of the unit square in $\mathbb{R}^{20}$ encode the ratio of the maximum amount of each military unit.
- This allied unit behavioral space was used to generate 4 different embeddings, 3 of them in $\mathbb{R}^2$ using PCA, Isomap and UMAP, and one of them in $\mathbb{R}^8$ in PCA in order to explain more than 85% of the variance.
- We illuminated these embeddings with the ratio of Marines, Cyclones, Reapers and Hellions that were built in replays, identifying how different methods resulted in different structures and separations of strategies.
- Applying the BRIL approach, a model was trained for each one of these embeddings by expanding the input to include the different behavioral features.
- After comparing these models to the baseline, we noticed that there was no significant increase on the test accuracy when adding the behavioral features to the input.
- We coupled all these models with an open source StarCraft 2 Terran bot called `sc2bot` in order to try these policies behavior in-game.
- We tested the ability of these models to express multiple behaviors by using 4 different cluster centers in-game. After this experiment, we can arguably conclude that we are able to modify the behavior of the models.

- We studied the separation between strategies in each embedding by analyzing how many units of each type were built. After these experiments, we noticed that each embedding's distribution of strategies translated into different separation of behaviors of the bot.
- The ability of each bot to properly act with a particular behavior seems to depend on how separated said behavior is from other ones. This would imply that, in order to properly express an strategy, embeddings that separate this strategy clearly would work best in the BRIL approach.

5.3. Future Work

We consider that there is still plenty of research to be done with respect to the BRIL approach. In particular, our research raises questions about the use of $\mathbb{R}^2$ as the embedding space. The use of higher dimensions could result in models that are better at expressing behaviors, with the disadvantage of trading off instructing visualizations such as the illuminations we performed.

Another area in which BRIL could be extended or improved, is in the design of behavior spaces. A learning process could be implemented in order to automatically extract the relevant behavior spaces in a dataset. Using, for example, Autoencoders [12] in order to extract context out of demonstrations would be an interesting approach.

In the realm of StarCraft 2, future work could focus on designing more behavior spaces, leveraging e.g. information about the opponent or about the actual score metrics the player had.

Another field in which we consider the BRIL approach to be useful is text generation. Since articles have defined topics, they define a natural behavior space. Neural network models could be trained to express e.g. the behavior of writing a piece about Mathematics, or a news article about politics.

A. Data Gathering, Model Training and Bot building

This appendix contains relevant information about the computational tools used when making this thesis, including a description of a scrapper tool written by the author.

Almost all of the data processing, model training and bot implementing made in this thesis was written and run in versions of Python posterior to 3.6, using libraries such as numpy, pandas, matplotlib, PyTorch, scikit-learn pymongo, pysc2 [38], python-sc2, sc2bot, sc2reaper and click, as well as many core libraries of Python such as json and glob.

A.1. Data Gathering

Data is the main ingredient in any Supervised and Unsupervised Machine Learning algorithm. With regards to the videogame StarCraft II, data was gathered using a combination of the recently developed python library pysc2 and the work of Huikai Wu et al. in [40], in which they created a dataset (called the MSC dataset) for supervised learning in StarCraft II.

Each time a game of StarCraft II is played, the main events and actions of it are stored in a `.SC2Replay` file. This file can be run using the engine of the video game in order to re-watch and analyze the game that was played. This tool is frequently used by professional and amateur players, in order to better understand the mistakes they did in the matches they lost, or in order to study what strategies and timings good players use. The MSC database was extracted by using pysc2's interface with the binary of the game, and parsing several `.SC2Replay` files, made available by Blizzard, the creator of StarCraft, and DeepMind [38].

Even though MSC contains plenty of information, we decided to implement our own tool. This tool is called sc2reaper, and it empowers the user with a template to extract exactly the data they want. It is inspired in the way pysc2 and MSC extract data from replays, and it adds freedom with respect to which data to store, and how to store it. In the rest of this section we will explain how `sc2reaper` works.

`sc2reaper` ingests a `.SC2Replay` file by running the simulation of the game. It uses pysc2 to start the replay in the binary of the game, and starts querying the state of the game every fixed amount of steps (which the end user can define). At each query, it is able to read

general information about the replay such as the map it was played on, the race and MMR[1] of each player and how long the match was. It is also able to extract granular information about each state of the game, including for example as the amount of resources the player has, the amount and types of allied units, visible enemy units, the actions taken since the last query and so on.

In particular, at each iteration sc2reaper consolidates three MongoDB documents into different collections in a database specified and created by the end user:

- A `state` document, which contains ids for localization (such as an id for the replay and the frame in which this document was consolidated) and information about resources, supply, allied units, allied units in progress (and their progress), visible enemy units and the upgrades the player has. This is all the necessary information to construct a state abstraction, as was made in this thesis and in previous work on learning macromanagement actions from replays [17].
- An `action` document, which contains the same ids as the `state` document, but also a list of all actions that were made since the last query. These actions include, but are not limited to, move and attack commands, camera movements and macro actions (such as the ones tackled by this thesis).
- A `score` document, with information such as the resource recollection rate and the amount of resources lost due to combat, alongside the same ids as the previous two documents.

These three documents are then put into collections `states`, `actions` and `scores` respectively.

For this thesis, we processed 7777 `.SC2Replay` files, consolidating them into a database in a local MongoDB instance. This collections were later postprocessed using pymongo, an interface between Python and MongoDB. The end result of this postprocessing consisted on several `.csv` files, one per replay.

A.2. Model Training

These state-action pairs were then split in three parts: training (60%), validation (30%) and test (10%). All neural networks described in chapter 4 were trained using PyTorch. No dropout was used, and we implemented early stopping, which consists on using the validation set to measure if the neural network is overfitting to the training data. We store the model only when the accuracy on the validation set has increased. Moreover, this networks were trained in a server running Ubuntu 16.04 and using CUDA with an NVidia Titan X GPU.

[1] MMR stands for Match-Making Ratio, a measure of how good a player is which is used in order to pair evenly-matched players on the ladder. If you're familiar with Chess, think of the ELO score.

A.3. Bot building

In order to test our approach in-game, we used an open source bot built by Niels Justesen. This bot, called sc2bot[2], was inspired by previous work done by the researcher on StarCraft (in particular, his modifications of UAlbertaBot [27]).

The bot is built by manager modules, each one of them covering a particular component of the strategy. These managers include:

- An Army manager, that attacks or defends by using simple hard-coded rules.
- An Assault manager, that measures the value of the enemy's army and compares it to the bot's own army. This measure is then used to activate the attack or defense mechanisms in Army manager.
- A Production manager, in which our models assess the state of the game through the bot's observations and make decisions on what to build and research.
- A Building manager, that manages the creation of units and buildings after getting tasks by the Production Manager.
- A Worker manager, that selects and manages workers on a micro level for building, repairing and scouting.

[2] https://github.com/njustesen/sc2bot

Bibliography

[1] Bishop, Christopher M.: *Pattern Recognition and Machine Learning*. Springer, 2006

[2] Buro, Michael: Real-time strategy games: a new AI research challenge. In: *Proceedings of the 18th international joint conference* (2003), p. 1534–1535. – ISSN 10450823

[3] Carlsson, Gunnar: *Topology and data*. 2009. – 255–308 p.. – ISBN 0011051000

[4] Chen, Jie ; Saad, Yousef ; Fang, Haw-Ren: Fast approximate kNN graph construction for high dimensional data via recursive lanczos bisection. In: *Journal of Machine Learning Research* 10 (2009), Nr. 2009, p. 1989–2012. – ISSN 1533–7928

[5] Cully, Antoine ; Clune, Jeff ; Tarapore, Danesh ; Mouret, Jean-Baptiste: Robots that can adapt like animals. In: *Nature* 521 (2015), May, p. 503 EP –

[6] Cybenko, G.: Approximation by Superpositions of a Sigmoidal Function. In: *Mathematics of Control, Signals, and Systems* (1989), Nr. 2, p. 303–314

[7] E. Bredon, Glen: *Topology and Geometry*. Vol. 139. 1993

[8] Enström, Olof ; Hagström, Fredrik ; Segerstedt, John ; Viberg, Fredrik ; Wartenberg, Arvid ; Weber Fors, David: *Clustering and Classification of Time Series in Real-Time Strategy Games*. 2019

[9] Erickson, Jeff: *Algorithms*. self-published, 2019. – algorithms.wtf

[10] Esteva, Andre ; Kuprel, Brett ; Novoa, Roberto A. ; Ko, Justin ; Swetter, Susan M. ; Blau, Helen M. ; Thrun, Sebastian: Dermatologist-level classification of skin cancer with deep neural networks. In: *Nature* 542 (2017), Nr. 7639, p. 115–118. – ISBN 0028–0836

[11] Fefferman, Charles ; Mitter, Sanjoy ; Narayanan, Hariharan: Testing the manifold hypothesis. In: *Journal of the American Mathematical Society* 29 (2016), Nr. 4, p. 983–1049. – ISSN 0894–0347

[12] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – http://www.deeplearningbook.org

[13] Hastie, Trevor ; Tibshirani, Robert ; Friedman, Jerome: *The elements of statistical learning: data mining, inference and prediction*. 2. Springer, 2009

[14] Hauberg, Søren: Only Bayes should learn a manifold (on the estimation of differential geometric structure from data). In: *Preprint on arXiv* (2018)

[15] Hornik, Kurt ; Stinchcombe, Maxwell ; White, Halbert: Multilayer Feedforward Networks are Universal Approximators. In: *Neural Networks* 2 (1989), p. 359–366

[16] Justesen, Niels ; González Duque, Miguel ; Cabarcas Jaramillo, Daniel ; Mouret, Jean-Baptiste ; Risi, Sebastian: Learning a Behavioral Repertoire from Demonstrations. In: *arXiv* (2019)

[17] Justesen, Niels ; Risi, Sebastian: Learning macromanagement in starcraft from replays using deep learning. In: *2017 IEEE Conference on Computational Intelligence and Games, CIG 2017* (2017), p. 162–169. ISBN 9781538632338

[18] Justesen, Niels ; Risi, Sebastian: Automated Curriculum Learning by Rewarding Temporally Rare Events. In: *IEEE Conference on Computatonal Intelligence and Games, CIG* 2018-Augus (2018). – ISBN 9781538643594

[19] Karras, Tero ; Laine, Samuli ; Aila, Timo: A Style-Based Generator Architecture for Generative Adversarial Networks. In: *CoRR* abs/1812.04948 (2018)

[20] Krizhevsky, Alex. ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: *Neural Information Processing Systems* 25 (2012)

[21] LeCun, Y. ; Bottou, L. ; Bengio, Y. ; Haffner, P.: Gradient-Based Learning Applied to Document Recognition. In: *Proceedings of the IEEE* (1998). – ISBN 0018–9219

[22] Lee, Dennis ; Tang, Haoran ; Zhang, Jeffrey O. ; Xu, Huazhe ; Darrell, Trevor ; Abbeel, Pieter: Modular Architecture for StarCraft II with Deep Reinforcement Learning. In: *arXiv* (2018)

[23] Lee, John M.: Introduction to Smooth Manifolds. (2000)

[24] McInnes, Leland ; Healy, John ; Melville, James: UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. (2018)

[25] Mnih, Volodymyr ; Silver, David ; Riedmiller, Martin: Playing Atari with Deep Reinforcement Learning. , p. 1–9

[26] Mouret, Jean-Baptiste ; Clune, Jeff: Illuminating search spaces by mapping elites. (2015), p. 1–15. ISBN 9781479999934

[27] Ontanon, Santiago ; Synnaeve, Gabriel ; Uriarte, Alberto ; Richoux, Florian ; Churchill, David ; Preuss, Mike: A survey of real-time strategy game AI research and competition in starcraft. In: *IEEE Transactions on Computational Intelligence and AI in Games* 5 (2013), Nr. 4, p. 293–311. – ISBN 1943–068X

[28] OpenAI. *OpenAI Five Benchmark: Results*. https://blog.openai.com/openai-five-benchmark-results/. 2018

[29] Radford, Alec ; Wu, Jeff ; Child, Rewon ; Luan, David ; Amodei, Dario ; Sutskever, Ilya: Language Models are Unsupervised Multitask Learners. (2019)

[30] Rasmussen, Carl E. ; Williams, Christopher K. I.: *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005. – ISBN 026218253X

[31] Schmidhuber, Jürgen: Deep learning in neural networks: An overview. In: *Neural Networks* 61 (2015), p. 85 – 117. – ISSN 0893–6080

[32] Silver, David ; Huang, Aja ; Maddison, Chris J. ; Guez, Arthur ; Sifre, Laurent ; van den Driessche, George ; Schrittwieser, Julian ; Antonoglou, Ioannis ; Panneershelvam, Veda ; Lanctot, Marc ; Dieleman, Sander ; Grewe, Dominik ; Nham, John ; Kalchbrenner, Nal ; Sutskever, Ilya ; Lillicrap, Timothy ; Leach, Madeleine ; Kavukcuoglu, Koray ; Graepel, Thore ; Hassabis, Demis: Mastering the Game of Go with Deep Neural Networks and Tree Search. In: *Nature* 529 (2016), Januar, Nr. 7587, p. 484–489

[33] Silver, David ; Schrittwieser, Julian ; Simonyan, Karen ; Antonoglou, Ioannis ; Huang, Aja ; Guez, Arthur ; Hubert, Thomas ; Baker, Lucas ; Lai, Matthew ; Bolton, Adrian ; Chen, Yutian ; Lillicrap, Timothy ; Hui, Fan ; Sifre, Laurent ; van den Driessche, George ; Graepel, Thore ; Hassabis, Demis: Mastering the game of Go without human knowledge. In: *Nature* 550 (2017), Oktober, p. 354–

[34] Sonoda, Sho ; Murata, Noboru: Neural Network with Unbounded Activations is Universal Approximator. In: *CoRR* abs/1505.03654 (2015)

[35] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018

[36] Tenenbaum, Joshua B. ; De Silva, Vin ; Langford, John C.: A Global Geometric Framework for Nonlinear Dimensionality Reduction. In: *Science Reports* 290 (2000), Nr. December, p. 151–180

[37] Vinyals, Oriol ; Babuschkin, Igor ; Chung, Junyoung ; Mathieu, Michael ; Jaderberg, Max ; Czarnecki, Wojciech M. ; Dudzik, Andrew ; Huang, Aja ; Georgiev, Petko ; Powell, Richard ; Ewalds, Timo ; Horgan, Dan ; Kroiss, Manuel ; Danihelka, Ivo ; Agapiou, John ; Oh, Junhyuk ; Dalibard, Valentin ; Choi, David ; Sifre, Laurent ; Sulsky, Yury ; Vezhnevets, Sasha ; Molloy, James ; Cai, Trevor ; Budden, David ; Paine, Tom ; Gulcehre, Caglar ; Wang, Ziyu ; Pfaff, Tobias ; Pohlen, Toby ; Wu, Yuhuai ; Yogatama, Dani ; Cohen, Julia ; McKinney, Katrina ; Smith, Oliver ; Schaul, Tom ; Lillicrap, Timothy ; Apps, Chris ; Kavukcuoglu, Koray ; Hassabis, Demis ; Silver, David. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/. 2019

[38] Vinyals, Oriol ; Ewalds, Timo ; Bartunov, Sergey ; Georgiev, Petko ; Vezhnevets, Alexander S. ; Yeo, Michelle ; Makhzani, Alireza ; Uttler, Heinrich ; Agapiou, John ; Schrittwieser, Julian ; Gaffney, Stephen ; Petersen, Stig ; Simonyan, Karen ; Schaul, Tom ; Van Hasselt, Hado ; Silver, David ; Lillicrap, Timothy ; Calderone, Deepmind K. ; Keet, Paul ; Brunasso, Anthony ; Lawrence, David ; Ekermo, Anders ; Repp, Jacob ; Blizzard, Rodney T.: StarCraft II: A New Challenge for Reinforcement Learning.

[39] Wasserman, Larry: *All of Statistics: A Concise Course in Statistical Inference*. Springer Publishing Company, Incorporated, 2010. – ISBN 1441923225, 9781441923226

[40] Wu, Huikai ; Zhang, Junge ; Huang, Kaiqi: MSC: A Dataset for Macro-Management in StarCraft II. In: *arXiv preprint arXiv:1710.03131* (2017)

[41] Zha, H ; Zhang, Z: Isometric Embedding and Continuum ISOMAP. In: *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)* (2003), p. 864–871. ISBN 1577351894